EE 36 DATA STRUCTURES AND ALGORITHMS

(Common to EEE, EIE & ICE)

Aim: To master the design and applications of linear, tree, and graph structures. To understand various algorithm design and analysis techniques.

UNIT I :LINEAR STRUCTURES

Abstract Data Types (ADT) – List ADT – array-based implementation – linked list implementation – cursor-based linked lists – doubly-linked lists – applications of lists –Stack ADT – Queue ADT – circular queue implementation – Applications of stacks and queues

UNIT II: TREE STRUCTURES

Need for non-linear structures – Tree ADT – tree traversals – left child right sibling data structures for general trees – Binary Tree ADT – expression trees – applications of trees – binary search tree ADT

UNIT III :BALANCED SEARCH TREES AND INDEXING

AVL trees – Binary Heaps – B-Tree – Hashing – Separate chaining – open addressing –Linear probing

UNIT IV: GRAPHS

Definitions – Topological sort – breadth-first traversal - shortest-path algorithms – minimum spanning tree – Prim's and Kruskal's algorithms – Depth-first traversal – biconnectivity – euler circuits – applications of graphs

UNIT V :ALGORITHM DESIGN AND ANALYSIS

Greedy algorithms – Divide and conquer – Dynamic programming – backtracking – branch and bound – Randomized algorithms – algorithm analysis – asymptotic notations – recurrences – NP-complete problems

TOTAL : 45 PERIODS

TEXT BOOKS

1. M. A. Weiss, "Data Structures and Algorithm Analysis in C", Pearson Education Asia, 2002.
2. ISRD Group, "Data Structures using C", Tata McGraw-Hill Publishing Company Ltd., 2006.

REFERENCES

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "Data Structures and Algorithms", Pearson Education, 1983.
2. R. F. Gilberg, B. A. Forouzan, "Data Structures: A Pseudocode approach with C", Second Edition, Thomson India Edition, 2005.
3. Sara Baase and A. Van Gelder, "Computer Algorithms", Third Edition, Pearson Education, 2000.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms", Second Edition, Prentice Hall of India Ltd, 2001.

==========

Unit – I

1.1 Introduction:

What is data structure?

"The way information is organized in the memory of a computer is called a **data structure"**.

(OR)

A data structure is a way of organizing data that considers not only the items stored, but also their relationship to each other. Advance knowledge about the relationship between data items allows designing of efficient algorithms for the manipulation of data.

Definition of data structures

- Many algorithms require that we use a proper representation of data to achieve efficiency.
- This representation and the operations that are allowed for it are called data structures.
- Each data structure allows insertion, access, deletion etc.

Why do we need data structures?

- Data structures allow us to achieve an important goal: component reuse
- Once each data structure has been implemented once, it can be used over and over again in various applications.

Common data structures are

- Stacks
- Queues
- Lists
- Trees
- Graphs
- Tables

1.2 Classification Of Data Structure:

Based on how the data items or operated it will classified into

1. **Primitive Data Stucture :** is one the data items are operated closest to the machine level instruction.

   Eg : int, char and double.

2. **Non-Primitive Data Structure :** is one that data items are not operated closest to machine level instruction.

   2.1. **Linear Data Structure :** In which the data items are stored in sequence order.

   Eg: Arrays, Lists, Stacks and Queues.

Operations performed on any linear structure:

1. Traversal – Processing each element in the list
2. Search – Finding the location of the element with a given value.
3. Insertion – Adding a new element to the list.
4. Deletion – Removing an element from the list.
5. Sorting – Arranging the elements in some type of order.

==========

6. Merging – Combining two lists into a single list.

2.2. **Non Linear Data Structure :** In which the order of data items is not presence.

Eg : Trees, Graphs.

Linear Data Structure

2. List
   a. Array
      i. One Dimensional
      ii. Multi-Dimensional
      iii. Dynamic Array
      iv. Matrix
         1. Sparse Matrix
   b. Linked List
      i. Single Linked List
      ii. Double Linked List
      iii. Circular Linked List
   c. Ordered List
      i. Stack
      ii. Queue
         1. Circular Queue
         2. Priority Queue
      iii. Deque
3. Dictionary (Associative Array)
   a. Hash Table

Non-Linear Data Structures

1. Graph
   a. Adjacency List
   b. Adjacency Matrix
   c. Spanning Tree
2. Tree
   a. M-Way Tree
      i. B-Tree
         1. 2-3-4 Tree
         2. B+ Tree
   b. Binary Tree
      i. Binary Search Tree
      ii. Self-Balancing Binary Search Tree
         1. AVL Tree
         2. Red-Black Tree
         3. Splay Tree
      iii. Heap
         1. Min Heap
         2. Max Heap
         3. Binary Heap
      iv. Parse Tree

An example of several common data s **Characteristics of Data Structures**

Data Structure	Advantages	Disadvantages
Array	Quick inserts Fast access if index known	Slow search Slow deletes Fixed size
Ordered Array	Faster search than unsorted array	Slow inserts Slow deletes Fixed size
Stack	Last-in, first-out acces	Slow access to other items
Queue	First-in, first-out access	Slow access to other items

Linked List	Quick inserts Quick deletes	Slow search
Binary Tree	Quick search Quick inserts Quick deletes *(If the tree remains balanced)*	Deletion algorithm is complex
Red-Black Tree	Quick search Quick inserts Quick deletes *(Tree always remains balanced)*	Complex to implement
2-3-4 Tree	Quick search Quick inserts Quick deletes *(Tree always remains balanced)* *(Similar trees good for disk storage)*	Complex to implement
Hash Table	Very fast access if key is known Quick inserts	Slow deletes Access slow if key is not known Inefficient memory usage
Heap	Quick inserts Quick deletes Access to largest item	Slow access to other items
Graph	Best models real-world situations	Some algorithms are slow and very complex

1.3 Abstract Data Types

Abstract data type (**ADT**) is a specification of a set of data and the set of operations that can be performed on the data.

Examples

- Associative array
- Set
- Stack
- Queue
- Tree

Uses of ADT: -

1. It helps to efficiently develop well designed program
2. Facilitates the decomposition of the complex task of developing a software system into a number of simpler subtasks
3. Helps to reduce the number of things the programmer has to keep in mind at any time
4. Breaking down a complex task into a number of earlier subtasks also simplifies testing and debugging

Algorithm:

Definition: An *algorithm* is a finite set of instructions which, if followed, accomplish a particular task. In addition every algorithm must satisfy the following criteria:

1. input: there are zero or more quantities which are externally supplied;
2. output: at least one quantity is produced;
3. definiteness: each instruction must be clear and unambiguous;
4. finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

1.4 Linear Data Structures

A data structure is said to be *linear* if its elements form a sequence or a linear list.

Examples:

Arrays
Linked Lists
Stacks, Queues

1.4.1 Arrays

- The very common linear structure is array. Since arrays are usually easy to traverse, search and sort, they are frequently used to store relatively permanent collections of data.
- An array is a list of a finite number n of *homogeneous* data elements (i.e., data elements of the same type) such that:
  a) The elements of the array are referenced respectively by an *index* consisting of n consecutive numbers.
  b) The elements of the array are stored respectively in successive memory locations.

Operations of Array:

- Two basic operations in an array are *storing* and *retrieving (extraction)*

Storing: A value is stored in an element of the array with the statement of the form,

Data[i] = X ; Where I is the valid index in the array
And X is the element

Extraction : Refers to getting the value of an element stored in an array.

X = Data [i], Where I is the valid index of the array and X is the element.

Array Representation:

- ✓ The number ***n*** of elements is called the *length* or *size* of the array. If not explicitly stated we will assume that the index starts from 0 and end with n-1.
- ✓ In general, the length (range) or the number of data elements of the array can be obtained from the index by the formula,

Length = UB – LB + 1

- ✓ Where UB is the largest index, called the Upper Bound, and LB is the smallest index, called Lower Bound, of the array.
- ✓ If LB = 0 and UB = 4 then the length is,

Length = 4 – 0 + 1 = 5

- ✓ The elements of an array A may be denoted by the subscript notation (or bracket notation), **A[0], A[1], A[2], … , A[N]**
- ✓ The number K in A[K] is called a *subscript* or an *index* and A[K] is called a *subscripted variable*.
- ✓ Subscripts allow any element of A to be referenced by its relative position in A.
- ✓ If each element in the array is referenced by a single subscript, it is called single dimensional array.
- ✓ In other words, the number of subscripts gives the dimension of that array.

Two-dimensional Arrays :

- ✓ A two-dimensional *mXn* array A is a collection of m*n data elements such that each element is specified by a pair of integers (such as I, J), called subscripts, with the property that, **0 ≤ I < m and 0 ≤ J < n**
- ✓ The element of A with first subscript ***i*** and second subscript ***j*** will be denoted by,

A[i,j] or A[i][j] *(c language)*

- ✓ Two-dimensional arrays are called *matrices* in mathematics and *tables* in business applications; hence two-dimensional arrays are sometimes are called *matrix arrays.*
- ✓ There is a standard way of drawing a two-dimensional *mXn* array A where the elements of A form a rectangular array with m rows and n columns and where the element A[i][j] appears in ***row i*** and ***column j***.
- ✓ A *row* is a horizontal list of elements, and a *column* is a vertical list of elements.

Example:

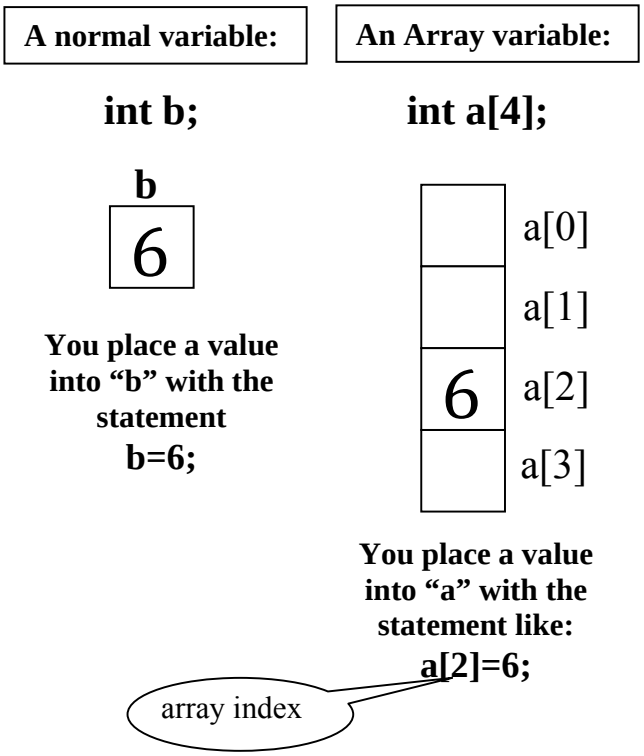
		Columns		
		0	1	2
	0	A[0][0]	A[0][1]	A[0][2]
Rows	1	A[1][0]	A[1][1]	A[1][2]
	2	A[2][0]	A[2][1]	A[2][2]

- ✓ The two-dimensional array will be represented in memory by a block of m*n sequential memory locations.
- ✓ Specifically, the programming languages will store the array either
  1. Column by column, i.e. ***column-major order,*** or
  2. Row by row, i.e. ***row-major order.***

Abstract Data Types (ADT)

- ❖ The ADT consists of a set of definitions that allow programmers to use the functions while hiding the implementation. This generalization of operations with unspecified implementations is known as abstraction.
- ❖ An ADT is a data declaration packaged together with the operations that are meaningful on the data type.
  1. Declaration of Data
  2. Declaration of Operations

An array is a collection of memory locations which allows storing homogeneous elements. It is an example for linear data structure.



An array lets you declare and work with a collection of values of the same type (homogeneous). For example, you might want to create a collection of five integers. One way to do it would be to declare five integers directly:

```
int a, b, c, d, e;
```

Suppose you need to fined average of 100 numbers. What will you do? You have to declare 100 variables. For example:

```
int a, b, c, d, e, f, g, h, i, j, k, l, m, n... etc.,
```

An easier way is to declare an array of 100 integers:

```
int a[100];
```

The General Syntax is:

datatype array_name [size];

Subscript

Example:

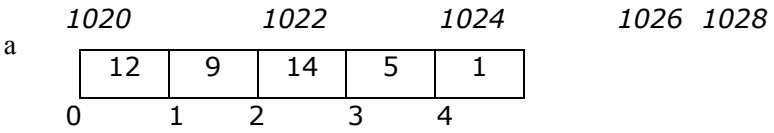
int a[5];

The five separate integers inside this array are accessed by an **index**. All arrays start at index zero and go to n-1 in C. Thus, **int a[5];** contains five elements. For example:

```
a[0] = 12;
a[1] = 9;
a[2] = 14;
a[3] = 5;
a[4] = 1;
```

Note: The array name will hold the address of the first element. It is called as BASE ADDRESS of that array. The base address can't be modified during execution, because it is static. It means that the increment /decrement operation would not work on the base address.

Consider the first element is stored in the address of 1020. It will look like this,



a[0] means a + 0 ➔ 1020 + 0 ➔ 1020 (locates the 1020)

a[1] means a + 1 ➔ 1020 + 1 * size of datatype ➔ 1020 + 2 ➔ 1022[for 'int' size is 2 byte]

a[2] means a + 2 ➔ 1020 + 2 * size of datatype ➔ 1020 + 4 ➔ 1024

a[3] means a + 3 ➔ 1020 + 3 * size of datatype ➔ 1020 + 6 ➔ 1026

a[4] means a + 4 ➔ 1020 + 4 * size of datatype ➔ 1020 + 8 ➔ 1028

Array indexing helps to manipulate the index using a for loop. Because of that retrieval of element from an array is very easy. For example, the following code ***initializes all of the values in the array to 0:***

```
int a[5]; /* Array declaration */
int i;

/* Initializing Array Elements to 0 */
for (i=0; i<5; i++)
   a[i] = 0;

/* print array */
printf("Elements in the array are…\n");
for (i=0; i < 5; i++)
          printf("%d\n",a[i]);
```

Note : (*mathematics*) A matrix most of whose entries are zeros.

Advantages:

- Reduces memory access time, because all the elements are stored sequentially. By incrementing the index, it is possible to access all the elements in an array.
- Reduces no. of variables in a program.
- Easy to use for the programmers.

Disadvantages:

- Wastage of memory space is possible. *For example: Storing only 10 elements in a 100 size array. Here, remaining 90 elements space is waste because these spaces can't be used by other programs till this program completes its execution.*
- Storing heterogeneous elements are not possible.
- Array bound checking is not available in 'C'. So, manually we have to do that.

Note: Memory representation of 1, 2 and multi-dimensional array refer class notes

1.4.1.1 Structures

Struct : Declares a structure, an object consisting of multiple data items that may be of different types.

DEFINING A STRUCTURE:

Syntax:

optional

```
struct tag
{
    data-type member 1;
    data-type member 2;
    ..........
    data-type member m;
};
```

Don't forget the Semicolon here

Here, *struct* is the required keyword; *tag* (optional) is a name that identifies structures of this type; and member1, meber2, …, member m are individual member declarations.

- The individual members can be ordinary variables, pointers, arrays, or other structures.
- A storage class cannot be assigned to an individual member, and individual members can not be initialized within a structure type declaration.

DECLARING STRUCTURE VARIABLES:

Once the composition of the structure has been defined, individual structure-type variables can be declared as follows:

storage-class struct tag variable1, varibale2, …, variable n;

where storage-class is an optional storage class specifier, struct is a required keyword, tag is the name that appeared in the structure declaration and variable1, variable2, …, variable n are structure variables of type tag.

Example:

```
struct student
{
        int regno;
        char name[20];
        char dept[10];
        int year;
};
```

Here, regno, name, dept and year are the members of the student structure. And this is the definition of the datatype. So, no memory will be allocated at this stage. The memory will be allocated after the declaration only. Structure variables can be declared as following methods:

a) Normal way of declaration

```
struct student s1, s2;
```

b) It is possible to combine the declaration of the structure composition with that of the structure variables, as shown below:

```
struct student
{
        int regno;
        char name[20];
        char dept[10];
        int year;
} s1, s2;
```

c) If we are going to declare all the necessary structure variables at definition time then we can create them without the tag, as shown below:

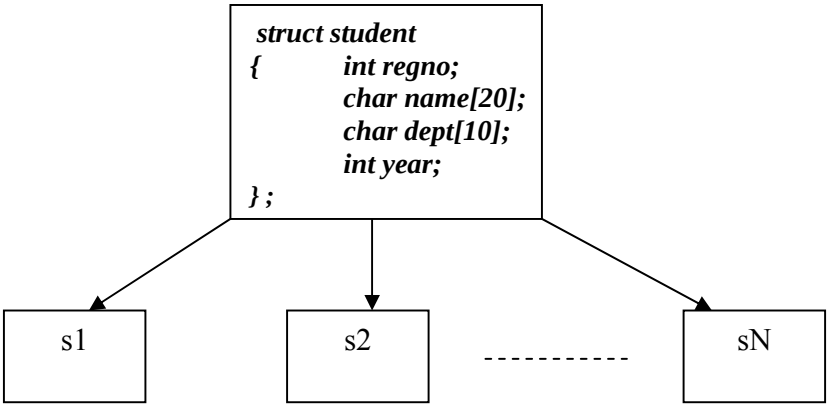
```
struct
{
        int regno;
        char name[20];
        char dept[10];
        int year;
} s1, s2;
```

Since there is no tag name, additional variables can not be generated other than this location. i.e. can't create new variables with this structure in the local functions. If we want we have to redefine the structure variable once again.

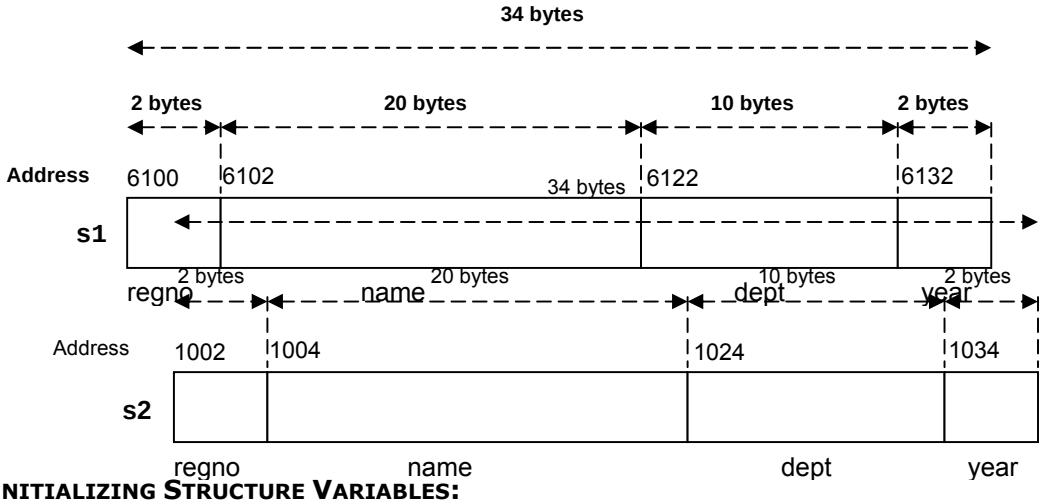
d) If we use the *typedef* in front of the *struct* keyword then the tag name alone can be used in other places whenever you want to use the student data type.

```
typedef struct student
{
```

```
    int regno;
    char name[20];
    char dept[10];
    int year;
} ;student s1, s2;   /* here the struct keyword is not needed because of typedef */
```



The size of each of these variables is 34 bytes because the size of the student datatype is 34 bytes. And the memory will be allocated for each variable as follows:



INITIALIZING STRUCTURE VARIABLES:

The members of a structure variable can be assigned initial values in much the same manner as the elements of an array. The initial values must appear in the order in which they will be assigned to their corresponding structure members, enclosed in braces and separated by commas.

The general form is,

storage-class struct tag variable = {value1, value2, …,value n};

A structure variable, like an array, can be initialized only if its storage class is either external or static.

Example:

```
static struct student s1 = { 340, "Kumara Vel", "CSE", 3};
```

static struct student s2 = {533, "Sankari", "CSE", 4};

STORING VALUES INTO THE MEMBERS OF THE STRUCTURE VARIABLES:

a) Values may be stored by assignment operation.

```
s1.regno = 500;
strcpy(s1.name, "Surya");
strcpy(s1.dept, "CSE");
s1.year = 3;
```

b) also the scanf statement may be used to give values through the keyboard.

```
scanf("%d", &s1.regno);
scanf("%s", s1.name);
scanf("%s", s1.dept);
scanf("%d", &s1.year);                    OR
scanf("%d%s%s%d", &s1.regno, s1.name, s1.dept, &s1.year);
```

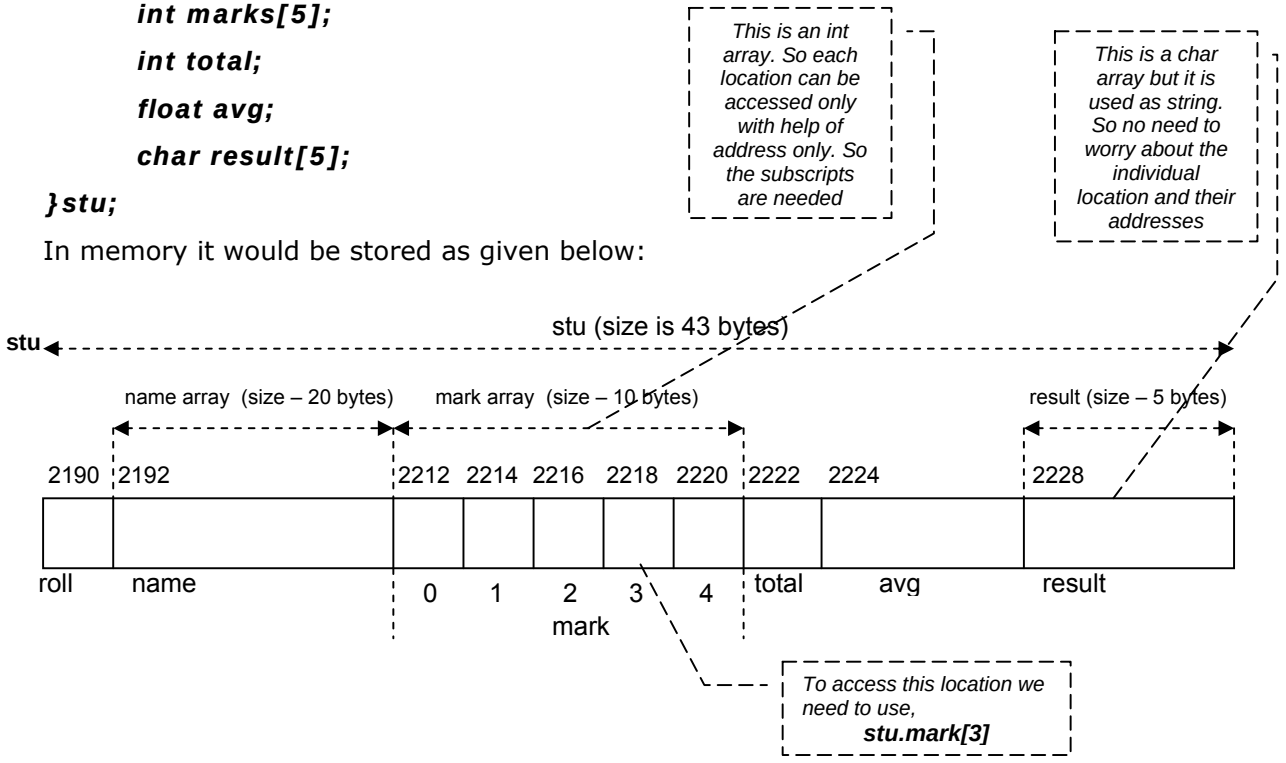
ARRAYS IN THE STRUCTURE:

The derived data types like array can be included in the structure as a member.

Example:

```
struct student
{
    int roll;
    char name[20];
    int marks[5];
    int total;
    float avg;
    char result[5];
}stu;
```

In memory it would be stored as given below:



NESTED STRUCTURES:

A structure variable may be defined as a member of another structure. In such situations, the declaration of the embedded structure must appear before the declaration of the outer structure.

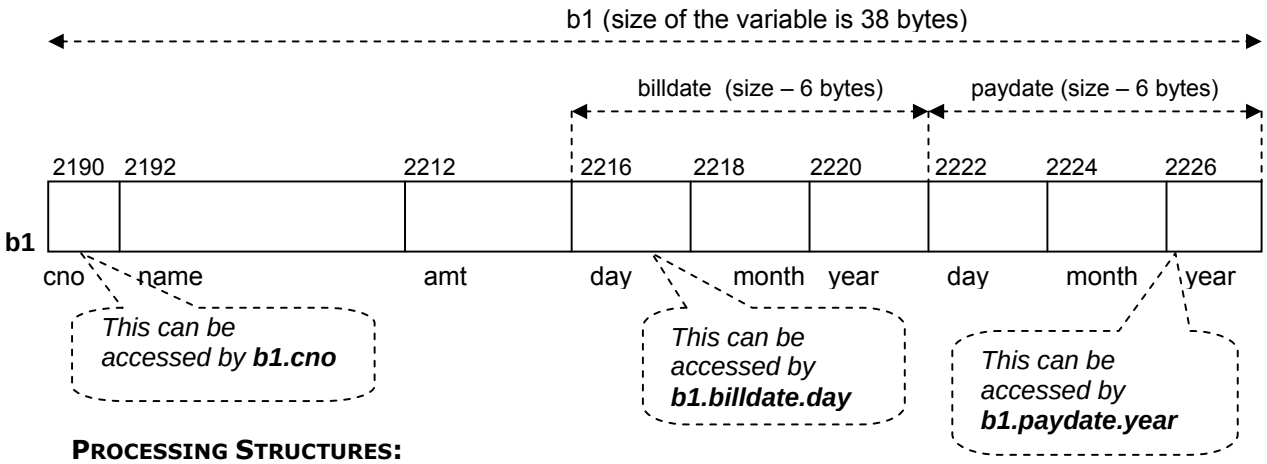
Example:

```
struct date
{
        int day;
        int month;
        int year;
};
struct bill
{
        int cno;
        char name[20];
        float amt;
        struct date billdate;
        struct date paydate;
}b1, b2;
```

OR

```
struct bill
{
        int cno;
        char name[20];
        float amt;
        struct date
        {
                int day;
                int month;
                int year;
        }billdate, paydate;

}b1, b2;
```

The second structure bill now contains another structure, date, as one of its members. The structure may look like as follows:



PROCESSING STRUCTURES:

Consider the following structure:

```
struct student
{
        int regno;
        char name[20];
        char dept[10];
        struct date
```

```
        {
                int day;
                int month;
                int year;
        }bday;
        int marks[5];
        int year;
} s1;
```

The members of a structure are usually processed individually, as separate entities. Therefore, we must be able to access the individual structure members. A structure member can be accessed by writing **structure_variable.member;**

where variable refers to the name of a structure-type variable, and member refers to the name of a member within the structure. The period (.) separates the variable name from the member name. It is a member of the highest precedence group, and its associativity is left to right.

Example ***s1.regno, s1.name, s1.dept, s1.year***

A nested structure member can be accessed by writing

structure_variable.member.submember;

Example ***s1.bday.day, s1.bday.month, s1.bday.year***

where member refers to the name of the member within the outer structure, and submember refers to the name of the member within the embedded structure. similarly, if a structure is an array, then an individual array element can be accessed by writing

structure-variable.member[expression];

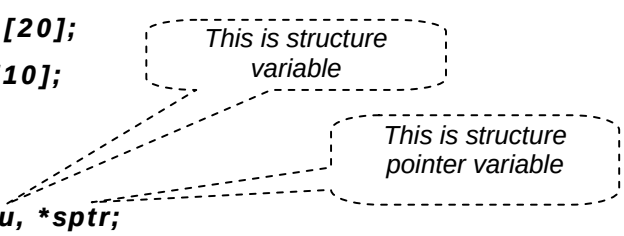
Example: ***s1.mark[0], s1.mark[1], s1.mark[2], s1.mark[3], s1.mark[4]***

POINTERS TO STRUCTURES:

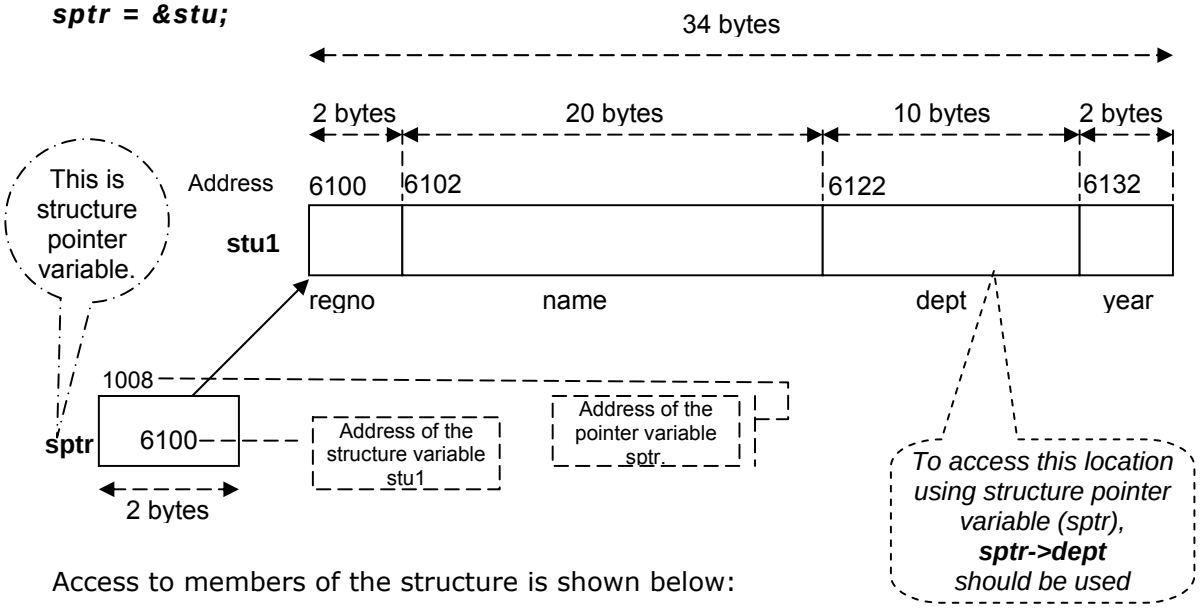
The address of a given structure variable can be obtained by using the & operator. Pointers to structures, like all other pointer variables may be assigned addresses. The following statements illustrate this concept.

Example:

```
struct student
{
        int regno;
        char name[20];
        char dept[10];
        int year;
};
struct student stu, *sptr;
```



sptr = &stu;



Access to members of the structure is shown below:

```
printf("Student Registration Number : %d\n", sptr->regno);
printf("Student Name                : %s\n", sptr->name);
printf("Department Name             : %s\n", sptr->dept);
printf("Year of Study               : %d\n", sptr->year);
```

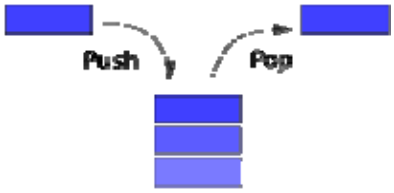
1.4.2 STACK :

"A *stack* is an ordered list in which all insertions and deletions are made at one end, called the *top".* stacks are sometimes referred to as Last In First Out (LIFO) lists

Stacks have some useful terminology associated with them:

- **Push** To add an element to the stack
- **Pop** To remove an element from the stock
- **Peek** To look at elements in the stack without removing them
- **LIFO** Refers to the last in, first out behavior of the stack
- **FILO** Equivalent to LIFO

STACK (DATA STRUCTURE)



Simple representation of a stack

Given a stack S=(a[1],a[2],.....a[n]) then we say that a1 is the bottom most element and element a[i]) is on top of element a[i-1], 1<i<=n.

Implementation of stack :

1. array (static memory).
2. linked list (dynamic memory)

The operations of stack is

1. PUSH operations
2. POP operations
3. PEEK operations

The Stack ADT

A stack S is an abstract data type (ADT) supporting the following three methods:

push(n): Inserts the item ***n*** at the top of stack

pop() : Removes the top element from the stack and returns that top element. An error occurs if the stack is empty.

peek(): Returns the top element and an error occurs if the stack is empty.

1. Adding an element into a stack. (called PUSH operations)

Adding element into the TOP of the stack is called PUSH operation.

Check conditions :

TOP = N , then **STACK FULL**

where N is maximum size of the stack.

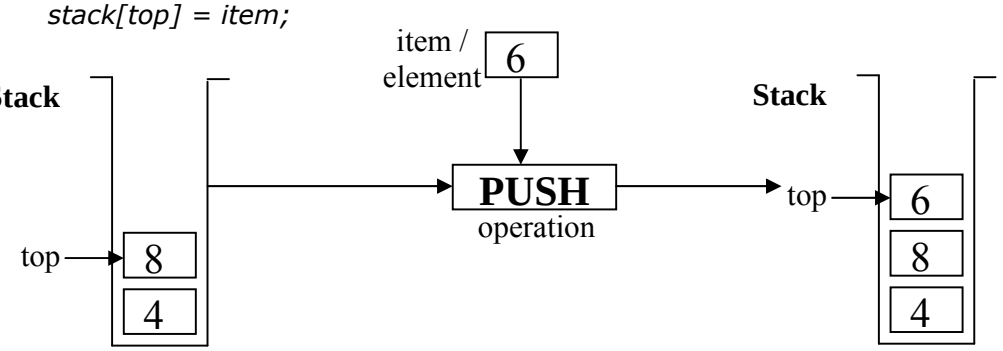
PUSH algorithm

```
procedure add(item : items);
{
add item to the global stack stack ;   top is the current top of stack
and n is its maximum size}
begin
   if top = n then stackfull;
   top := top+1;
   stack(top) := item;
end: {of add}
```

Implementation in C using array:

```
/* here, the variables stack, top and size are global variables */
 void push (int item)
{
        if (top == size-1)
                printf("Stack is Overflow");
        else
        {
                top = top + 1;
```

```
        stack[top] = item;
    }
}
```



2. Deleting an element from a stack. (called POP operations)

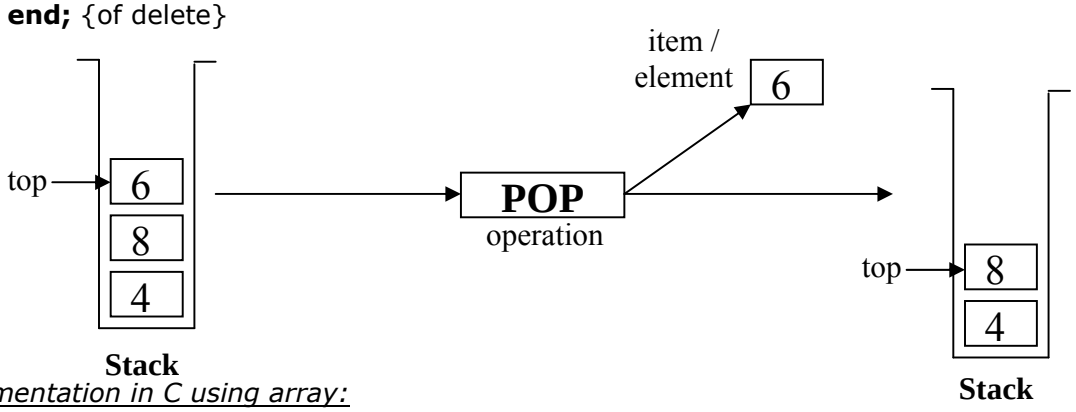
Deleting or Removing element from the TOP of the stack is called POP operations.

Check Condition:

TOP = 0**, then** STACK EMPTY

Deletion in stack (POP Operation)

```
procedure delete(var item : items);
{
    remove top element from the stack stack and put it in the item}
    begin
      if top = 0 then stackempty;
      item := stack(top);
      top := top-1;
    end; {of delete}
```



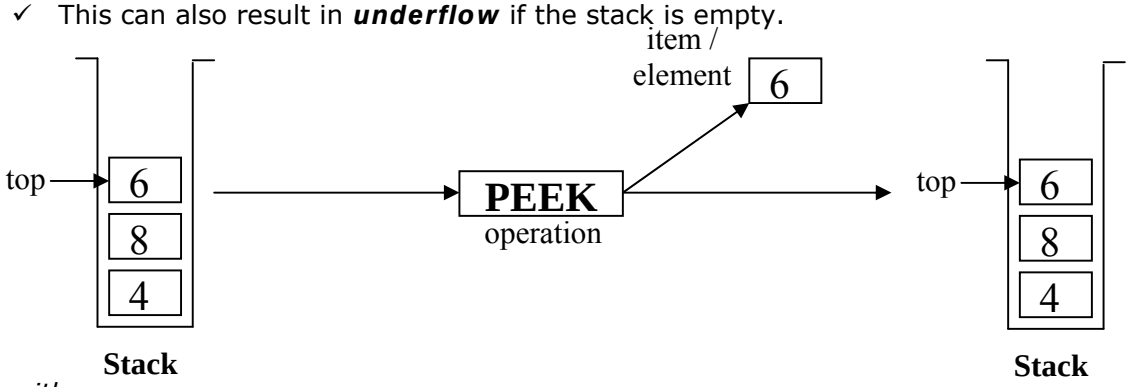
Implementation in C using array:

```
/* here, the variables stack, and top are global variables */
 int pop ( )
{
    if (top == -1)
    {
        printf("Stack is Underflow");
```

```
        return (0);
    }
    else
    {
        return (stack[top--]);
    }
}
```

3. Peek Operation:

- ✓ Returns the item at the top of the stack but does not delete it.
- ✓ This can also result in ***underflow*** if the stack is empty.



Algorithm:

PEEK(STACK, TOP)

```
BEGIN
    /* Check, Stack is empty? */
    if (TOP == -1) then
        print "Underflow" and return 0.
    else
        item = STACK[TOP]  / * stores the top element into a local variable */
        return item        / * returns the top element to the user */
END
```

Implementation in C using array:

```
/* here, the variables stack, and top are global variables */
 int pop ( )
{
    if (top == -1)
    {
        printf("Stack is Underflow");
        return (0);
    }
    else
```

```
        {
                return (stack[top]);
        }
}
```

Applications of Stack

1. It is very useful to evaluate arithmetic expressions. (Postfix Expressions)
2. Infix to Postfix Transformation
3. It is useful during the execution of recursive programs
4. A Stack is useful for designing the compiler in operating system to store local variables inside a function block.
5. A stack (memory stack) can be used in function calls including recursion.
6. Reversing Data
7. Reverse a List
8. Convert Decimal to Binary
9. Parsing – It is a logic that breaks into independent pieces for further processing
10. Backtracking

Examples :

1. Infix notation A+(B*C) equivalent Postfix notation ABC*+
2. Infix notation (A+B)*C Equivalent Postfix notation AB+C*

Expression evaluation and syntax parsing

Calculators employing reverse Polish notation (also known as **postfix notation**)use a stack structure to hold values.

Expressions can be represented in prefix, postfix or infix notations. Conversion from one form of the expression to another form needs a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code. Most of the programming languages are context-free languages allowing them to be parsed with stack based machines. Note that natural languages are context sensitive languages and stacks alone are not enough to interpret their meaning.

Infix, Prefix and Postfix Notation:

We are accustomed to write arithmetic expressions with the operation between the two operands: **a+b** or **c/d**. If we write **a+b*c**, however, we have to apply precedence rules to avoid the ambiguous evaluation (add first or multiply first?).

There's no real reason to put the operation between the variables or values. They can just as well precede or follow the operands. You should note the advantage of prefix and postfix: the need for precedence rules and parentheses are eliminated.

Infix	Prefix	Postfix
a + b	+ a b	a b +
a + b * c	+ a * b c	a b c * +
(a + b) * (c - d)	* + a b - c d	a b + c d - *
b * b - 4 * a * c		
40 - 3 * 5 + 1		

Examples of use: (application of stack)

Arithmetic Expressions: Polish Notation

- An arithmetic expression will have operands and operators.
- Operator precedence listed below:

  Highest : ($)

  Next Highest : (*) and (/)

  Lowest : (+) and (-)
- For most common arithmetic operations, the operator symbol is placed in between its two operands. This is called ***infix notation***.

  *Example: A + B , E * F*
- Parentheses can be used to group the operations.

  *Example: (A + B) * C*
- Accordingly, the order of the operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.
- Polish notation refers to the notation in which the operator symbol is placed before its two operands. This is called ***prefix notation***.

  *Example: +AB, *EF*
- The fundamental property of polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression.
- Accordingly, one never needs parentheses when writing expressions in Polish notation.
- ***Reverse Polish Notation*** refers to the analogous notation in which the operator symbol is placed after its two operands. This is called ***postfix notation***.

  *Example: AB+, EF**
- Here also the parentheses are not needed to determine the order of the operations.
- *The computer usually evaluates an arithmetic expression written in infix notation in two steps,*

  1. *It converts the expression to **postfix notation**.*

2. *It evaluates the postfix expression.*

- *In each step, the stack is the main tool that is used to accomplish the given task.*

(1)Question : (Postfix evaluation)

How to evaluate a mathematical expression using a stack The algorithm for Evaluating a postfix expression ?

- Initialise an empty stack
- While token remain in the input stream
  - Read next token
  - If token is a number, push it into the stack
  - Else, if token is an operator, pop top two tokens off the stack, apply the operator, and push the answer back into the stack
- Pop the answer off the stack.

Algorithm postfixexpression

```
Initialize a stack, opndstk to be empty.
{
        scan the input string reading one element at a time into symb
}
While ( not end of input string )
{
Symb := next input character;
        If symb is an operand Then
                push (opndstk,symb)
        Else
                [symbol is an operator]
        {
                Opnd1:=pop(opndstk);
                Opnd2:=pop(opndnstk);
                Value := result of applying symb to opnd1 & opnd2
                Push(opndstk,value);
}
Result := pop (opndstk);
```

Example:

6 2 3 + - 3 8 2 / + * 2 $ 3 +

Symbol	*Operand 1 (A)*	*Operand 2 (B)*	*Value (A ⊗ B)*	*STACK*
6				6

2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3				1, 3
8				1, 3, 8
2				1, 3, 8, 2
/	8	2	/	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2				7, 2
$	7	2	49	49
3				49, 3
+	49	3	52	***52***

The Final value in the STACK is 52. This is the answer for the given expression.

(2) run time stack for function calls (write factorial number calculation procedure)

push local data and return address onto stack

return by popping off local data and then popping off address and returning to it

return value can be pushed onto stack before returning, popped off by caller

(3) expression parsing

e.g. matching brackets: [... (... (...) [...(...) ...] ...) ...]

push left ones, pop off and compare with right ones

4) Infix To Postfix Conversion

Infix expressions are often translated into postfix form in which the operators appear after their operands. **Steps:**

1. Initialize an empty stack.
2. Scan the Infix Expression from left to right.
3. If the scannned character is an operand, add it to the Postfix Expression.
4. If the scanned character is an operator and if the stack is empty, then push the character to stack.
5. If the scanned character is an operator and the stack is not empty, Then
   (a) Compare the precedence of the character with the operator on the top of the stack.
   (b) While operator at top of stack has higher precedence over the scanned character & stack is not empty.
      (i) POP the stack.

(ii) Add the Popped character to Postfix String.

(c) Push the scanned character to stack.

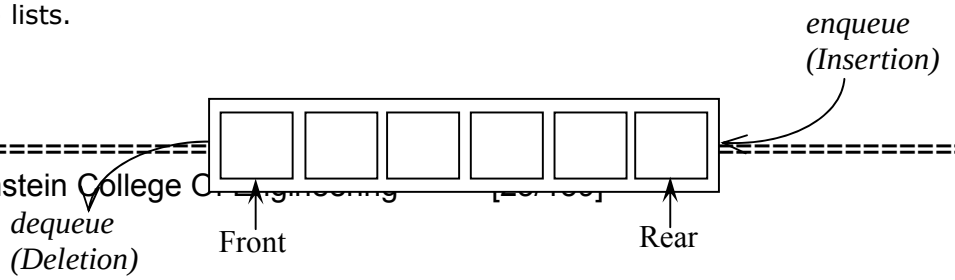
6. Repeat the steps 3-5 till all the characters
7. While stack is not empty,
   (a) Add operator in top of stack
   (b) Pop the stack.
8. Return the Postfix string.

Algorithm Infix to Postfix conversion (without parenthesis)

```
1. Opstk = the empty stack;
2. while ( not end of input )
   {
   symb = next input character;
3.  if ( symb is an operand )
          add symb to the Postfix String
4. else
    {
5.   While( ! empty (opstk) && prec ( stacktop ( opstk), symb) )
        {
         topsymb = pop ( opstk )
         add topsymb to the Postfix String;
         }  / * end of while */
      Push(opstk, symb);
       } /* end  else */
6.       }  /*  end while * /
7. While( ! empty ( opstk ) )
         {
         topsymb = pop (opstk)
         add topsymb to the Postfix String
         }  /*  end of while */
8. Return the Postfix String.
```

1.4.3 QUEUE :

"A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT". *queue*s are sometimes referred to as First In First Out (FIFO) lists.



Example

1. The people waiting in line at a bank cash counter form a queue.
2. In computer, the jobs waiting in line to use the processor for execution. This queue is called ***Job Queue***.

Operations Of Queue

There are two basic queue operations. They are,

Enqueue – Inserts an item / element at the rear end of the queue. An error occurs if the queue is full.

Dequeue – Removes an item / element from the front end of the queue, and returns it to the user. An error occurs if the queue is empty.

1. Addition into a queue

```
procedure addq (item : items);
{add item to the queue q}
begin
   if rear=n then queuefull
   else begin
       rear :=rear+1;
       q[rear]:=item;
   end;
end;{of addq}
```

2. Deletion in a queue

```
procedure deleteq (var item : items);
{delete from the front of q and put into item}
begin
   if front = rear then queueempty
   else begin
       front := front+1
       item := q[front];
```

```
        end;
    end
```

Uses of Queues (Application of queue)

Queues remember things in first-in-first-out (FIFO) order. Good for fair (first come first served) ordering of actions.

Examples of use: (Application of stack)

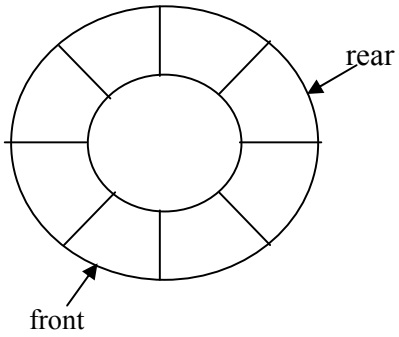
1• scheduling
processing of GUI events
printing request

2• simulation
orders the events
models real life queues (e.g. supermarkets checkout, phone calls on hold)

Circular Queue :

Location of queue are viewed in a circular form. The first location is viewed after the last one.

Overflow occurs when all the locations are filled.



Algorithm Circular Queue Insert

```
Void  CQInsert ( int queue[ ], front, rear, item)
{
    if ( front = = 0 )
        front = front +1;
   if ( ( ( rear = maxsize ) && ( front = = 1 ) )  || ( ( rear ! = 0 ) && ( front = rear +1)))
    {
        printf( " queue overflow ");
        if( rear = = maxsize )
            rear = 1;
        else
            rear = rear + 1;
            q [ rear ] = item;
```

```
        }
}
```

Algorithm Circular Queue Delete

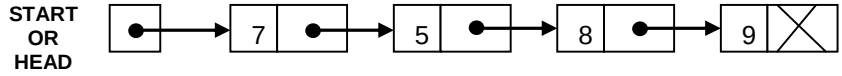
```
int CQDelete ( queue [ ], front, rear )
{
        if ( front = = 0 )
                printf ( " queue underflow ");
        else
          {
           item = queue [ front ];
           if(front = = rear )
                {
                front = 0; rear = 0;
                }
        else if ( front = = maxsize )
                {
                front = 1;
                }
        else
                front = front + 1;
         }
        return item;
}
```

1.4.4 Linked List

- Some demerits of array, leads us to use linked list to store the list of items. They are,
  1. It is relatively expensive to insert and delete elements in an array.
  2. Array usually occupies a block of memory space, one cannot simply double or triple the size of an array when additional space is required. *(For this reason, arrays are called **"dense lists"** and are said to be **"static"** data structures.)*
- A ***linked list***, or ***one-way list***, is a linear collection of data elements, called ***nodes***, where the linear order is given by means of ***pointers***. That is, each node is divided into two parts:
  - ✓ The first part contains the information of the element i.e. INFO or DATA.
  - ✓ The second part contains the ***link field***, which contains the address of the next node in the list.

- The linked list consists of series of nodes, which are not necessarily adjacent in memory.
- A list is a ***dynamic data structure*** i.e. the number of nodes on a list may vary dramatically as elements are inserted and removed.
- The pointer of the last node contains a special value, called the ***null*** pointer, which is any invalid address. This ***null pointer*** signals the end of list.
- The list with no nodes on it is called the ***empty list*** or ***null list***.

Example*: The linked list with 4 nodes.*



Types of Linked Lists:

a) Linear Singly Linked List
b) Circular Linked List
c) Two-way or doubly linked lists
d) Circular doubly linked lists

Advantages of Linked List

1. Linked List is dynamic data structure; the size of a list can grow or shrink during the program execution. So, maximum size need not be known in advance.
2. The Linked List does not waste memory
3. It is not necessary to specify the size of the list, as in the case of arrays.
4. Linked List provides the flexibility in allowing the items to be rearranged.

What are the pitfalls encountered in single linked list?

1. A singly linked list allows traversal of the list in only one direction.
2. Deleting a node from a list requires keeping track of the previous node, that is, the node whose link points to the node to be deleted.
3. If the link in any node gets corrupted, the remaining nodes of the list become unusable.

Linearly-linked List:

Is a collection of elements called Nodes. Each node consist of two fields, namely data field to hold the values and link(next) field points to the next node in the list.

It consists of a sequence of nodes, each containing arbitrary data fields and one or two references ("links") pointing to the next and/or previous nodes.

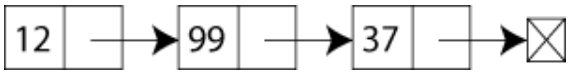
A linked list is a self-referential datatype (or) data structure because it contains a pointer or link to another data of the same type.

Linked lists permit insertion and removal of nodes at any point in the list in constant time, **but do not allow random access**.

Several different types of linked list exist: **singly-linked lists, doubly-linked lists, and circularly-linked lists.** One of the biggest advantages of linked lists is that nodes may have multiple pointers to other nodes, allowing the same nodes to simultaneously appear in different orders in several linked lists

Singly-linked list:

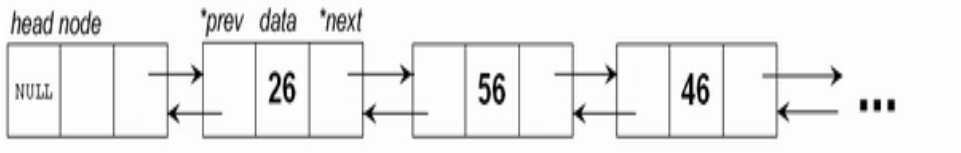
The simplest kind of linked list is a **singly-linked list** (or **slist** for short), which has one link per node. This link points to the next node in the list, or to a null value or empty list if it is the final node.



A singly linked list containing three integer values

Doubly-linked list:

A more sophisticated kind of linked list is a **doubly-linked list** or **two-way linked list**. Each node has two links: one points to the previous node, or points to a null value or empty list if it is the first node; and one points to the next, or points to a null value or empty list if it is the final node.



An example of a doubly linked list.

Circularly-linked list:

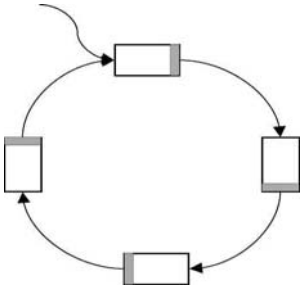
In a **circularly-linked list**, the first and final nodes are linked together. This can be done for both singly and doubly linked lists. To traverse a circular linked list, you begin at any node and follow the list in either direction until you return to the original node. Viewed another way, circularly-linked lists can be seen as having no beginning or end. This type of list is most useful for managing buffers for data ingest, and in cases where you have one object in a list and wish to see all other objects in the list.

The pointer pointing to the whole list is usually called the end pointer.

Singly-circularly-linked list:

In a **singly-circularly-linked list**, each node has one link, similar to an ordinary *singly-linked list*, except that the next link of the last node points back to the first node.

As in a singly-linked list, new nodes can only be efficiently inserted after a node we already have a reference to. For this reason, it's usual to retain a reference to only the last element in a singly-circularly-linked list, as this allows quick insertion at the beginning, and also allows access to the first node through the last node's next pointer.



- Note that there is no **NULL** terminating pointer
- Choice of **head** node is arbitrary
- A **tail** pointer serves no purpose
- *What purpose(s) does the* **head** *pointer serve?*

Doubly-circularly-linked list

In a **doubly-circularly-linked list**, each node has two links, similar to a *doubly-linked list*, except that the previous link of the first node points to the last node and the next link of the last node points to the first node. As in doubly-linked lists, insertions and removals can be done at any point with access to any nearby node.

Sentinel nodes

Linked lists sometimes have a special *dummy* or *sentinel node* at the beginning and/or at the end of the list, which is not used to store data.

Basic Operations on Linked Lists

1. Insertion
    a. At first
    b. At last
    c. At a given location (At middle)
2. Deletion
    a. First Node
    b. Last Node
    c. Node in given location or having given data item

Initial Condition

HEAD = NULL;

/* Address of the first node in the list is stored in HEAD. Initially there is no node in the list. So, HEAD is initialized to NULL (No address) */

What are the Applications of linked list?

- To implement of Stack, Queue, Tree, Graph etc.,
- Used by the Memory Manager

- To maintain Free-Storage List

Doubly Linked Lists (or) Two – Way Lists

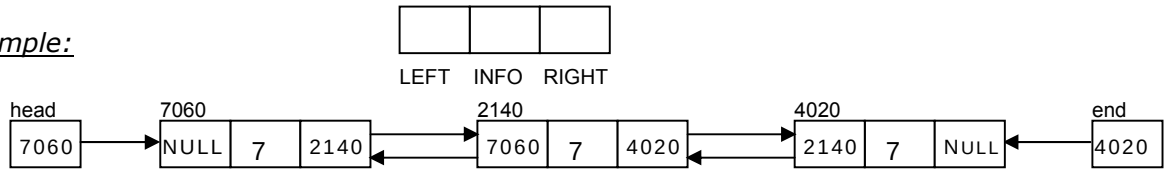
There are some problems in using the Single linked list. They are

1. A singly linked list allows traversal of the list in only one direction. (Forward only)
2. Deleting a node from a list requires keeping track of the previous node, that is, the node whose link points to the node to be deleted.

These major drawbacks can be avoided by using the double linked list. The doubly linked list is a ***linear collection*** of data elements, called ***nodes***, where each node is divided into three parts. They are:

1. *A pointer field LEFT which contains the address of the preceding node in the list*
2. *An information field INFO which contains the data of the Node*
3. *A pointer field RIGHT which contains the address of the next node in the list*

Example:



Linked lists vs. arrays

	Array	Linked list
Indexing	O(1)	O(*n*)
Inserting / Deleting at end	O(1)	O(1)
Inserting / Deleting in middle (with iterator)	O(*n*)	O(1)
Persistent	No	Singly yes
Locality	Great	Bad

Array	Linked list
Static memory	Dynamic memory
Insertion and deletion required to modify the existing element location	Insertion and deletion are made easy.
Elements stored as contiguous memory as on block.	Element stored as Non-contiguous memory as pointers
Accessing element is fast	Accessing element is slow

SINGLY LINKED LISTS

1. Insertion of a Node in the Beginning of a List

Step 1 : Allocate memory for a node and assign its address to the variable ' New'

Step 2 : Assign the element in the data field of the new node.

Step 3 : Make the next field of the new node as the beginning of the existing list.

Step 4 : Make the new node as the Head of the list after insertion.

Algorithm InsertBegin (Head, Elt)

[Adding the element **elt** in the beginning of the list pointed by Head]

1. new ← getnode (NODE)
2. data (new) ← elt
3. next (new) ← Head
4. Head ← new
5. return Head

Insertion of a Node at the End of a Singly Linked List

Step 1 : Allocate memory for a node and assign its address to the variable ' New'

Step 2 : Assign the element in the data field of the new node.

Step 3 : Make the next field of the new node as NULL This is because the new node will be the end of the resultant list.

Step 4 : If the existing list is empty, call this new node as the list. Else, get the address of the last node in the list by traversing from the beginning pointer.

Step 5: Make the next field of the last node point to the new node.

Algorithm InsertEnd (Head, Elt)

[Adding the element **elt** at the end of the list]

1. new ← getnode (NODE)
2. data (new) ← elt
3. next (new) ← NULL
4. if (Head == NULL) Then
   Head ← new
   Return Head
   Else
   Temp ← Head
5. While (next (temp) # NULL)
   temp ← next (temp)
6. next (temp) ← new
7. return Head.

Applications of linked lists

Linked lists are used as a building block for many other data structures, such as stacks, queues and their variations.

1. Polynomial ADT:

A polynomial can be represented with primitive data structures. For example, a polynomial represented as $a_k x^k$ $a_{k-1}x^{k-1}$ + ... + a_0 can be represented as a linked list. Each node is a structure with two values: a_i and i. Thus, the length of the list will be k. The first node will have (a_k, k), the second node will have (a_{k-1}, k-1) etc. The last node will be (a_0, 0).

The polynomial $3x^9 + 7x^3 + 5$ can be represented in a list as follows: (3,9) --> (7,3) --> (5,0) where each pair of integers represent a node, and the arrow represents a link to its neighbouring node.

Derivatives of polynomials can be easily computed by proceeding node by node. In our previous example the list after computing the derivative would represented as follows: (27,8) --> (21,2). The specific polynomial ADT will define various operations, such as multiplication, addition, subtraction, derivative, integration etc. A polynomial ADT can be useful for symbolic computation as well.

2. Large Integer ADT:

Large integers can also be implemented with primitive data structures. To conform to our previous example, consider a large integer represented as a linked list. If we represent the integer as successive powers of 10, where the power of 10 increments by 3 and the coefficient is a three digit number, we can make computations on such numbers easier. For example, we can represent a very large number as follows:

$513(10^6) + 899(10^3) + 722(10^0)$.

Using this notation, the number can be represented as follows:

(513) --> (899) --> (722).

The first number represents the coefficient of the 10^6 term, the next number represents the coefficient of the 10^3 term and so on. The arrows represent links to adjacent nodes.

The specific ADT will define operations on this representation, such as addition, subtraction, multiplication, division, comparison, copy etc.

An array allocates memory for all its elements lumped together as one block of memory. In contrast, a linked list allocates space for each element separately in its own block of memory called a "linked list element" or "node". The list gets is overall structure by using pointers to connect all its nodes together like the links in a chain.

Each node contains two fields: a "data" field to store whatever element type the list holds for its client, and a "next" field which is a pointer used to link one node to the next node.

Each node is allocated in the heap with a call to malloc(), so the node memory continues to exist until it is explicitly deallocated with a call to free(). The front of the list is a pointer to the first node. Here is what a list containing the numbers 1, 2, and 3 might look like...

malloc() malloc() is a system function which allocates a block of memory in the "heap" and returns a pointer to the new block. The prototype for malloc() and other heap functions are in stdlib.h. The argument to malloc() is the integer size of the block in bytes. Unlike local ("stack") variables, heap memory is not automatically deallocated when the creating function exits. malloc() returns NULL if it cannot fulfill the request. (extra for experts) You may check for the NULL case with assert() if you wish just to be safe. Most modern programming systems will throw an exception or do some other automatic error handling in their memory allocator, so it is becoming less common that source code needs to explicitly check for allocation failures.

free() free() is the opposite of malloc(). Call free() on a block of heap memory to indicate to the system that you are done with it. The argument to free() is a pointer to a block of memory in the heap — a pointer which some time earlier was obtained via a call to malloc().

Linked List implementation

In C: typically individual cells dynamically allocated containing a pointer to the next cell.

Advantages:

Space used adapts to size • usually results in better space usage than sequential despite storing a pointer in each cell

speed improvements for some operations

Disadvantages:

• speed reductions for some operations

Doubly Linked list

Allows efficient backwards traversal takes longer to insert and delete (but the same complexity) takes more space for the extra pointer (unless we use the for trick to save space at the cost of time)

Circular list (head and tail linked)

Two Marks

1. Limitations of arrays

e) Arrays have a fixed dimension. Once the size of an array is decided it cannot be increased or decreased during execution.

f) Array elements are always stored in contiguous memory locations. So it need contiguous locations otherwise memory will not be allocated to the arrays

g) Operations like insertion of a new element in an array or deletion of an existing element from the array are pretty tedious. This is because during insertion or deletion each element after the specified position has to be shifted one position to the right or one position to the left.

2. Define Data Structure.

A data structure is a way of organizing data that considers not only the items stored, but also their relationship to each other. Advance knowledge about the relationship between data items allows designing of efficient algorithms for the manipulation of data.

3. Why do we need data structures?

- Data structures allow us to achieve an important goal: ***component reuse***
- Once each data structure has been implemented once, it can be used over and over again in various applications.

4. Simple Classification of Data Structure.

The data structure can be classified into two types as follows:

a) Linear Data Structures – All the elements are formed in a sequence or maintain a linear ordering
   i. Arrays
   ii. Linked Lists
   iii. Stacks
   iv. Queues

b) Non-Linear Data Structures – All the elements are distributed on a plane i.e. these have no such sequence of elements as in case of linear data structure.
   i. Trees
   ii. Graphs
   iii. Sets

5. List the operations performed in the Linear Data Structure

a) Traversal – Processing each element in the list
b) Search – Finding the location of the element with a given value.
c) Insertion / Storing – Adding a new element to the list.
d) Deletion – Removing an element from the list.
e) Sorting – Arranging the elements in some type of order.
f) Merging – Combining two lists into a single list.

6. Explain Linked List

- A linked list is a list of elements in which the elements of the list can be placed anywhere in memory, and these elements are linked with each other using an explicit link field, that is by storing the address of the next element in the link field of the previous element.

- A linked list is a self-referential data type because it contains a pointer or link to another data of the same type. This permit insertion and removal of nodes at any point in the list in constant time, but do not allow random access.

7. What is a node?

Each element structure in a slinked list called node, containing two fields one is data and another is address of next data.

DATA	ADDRESS

8. Advantages of Linked List

5. Linked List is dynamic data structure; the size of a list can grow or shrink during the program execution. So, maximum size need not be known in advance.
6. The Linked List does not waste memory
7. It is not necessary to specify the size of the list, as in the case of arrays.
8. Linked List provides the flexibility in allowing the items to be rearranged.

9. What are the pitfalls encountered in single linked list?

4. A singly linked list allows traversal of the list in only one direction.
5. Deleting a node from a list requires keeping track of the previous node, that is, the node whose link points to the node to be deleted.
6. If the link in any node gets corrupted, the remaining nodes of the list become unusable.

10.Define Stack

- Stack is a linear data structure and is an ordered collection of homogeneous data elements, where the insertion and deletion operations takes place at one end called top of the stack.
- A stack data structure exhibits the LIFO (Last In First Out) property.

11.What are operations allowed in a Stack?

1. PUSH : This operation is used to add an item into the top of the stack.
2. POP : This operation is used to remove an item from the top of the stack.
3. PEEK : This operation is used to display the top item in the stack.

12.List the notations used to represent the arithmetic expressions.

1. Infix: <operand> operator <operand>
   Ex: A + B
2. Prefix: operator <operand> <operand>

(also called as polish notation) ***Ex: +AB***

3. Postfix: <operand> <operand> operator
   Ex: AB+

13.Rules for converting an Infix notation to postfix form

1. Assume, the fully parenthesized version of the Infix expression
2. Move all operator, so that they replace their corresponding right part of parenthesis
3. Remove all parenthesis

Example: ((A+((B^C)-D))*(E-(A/C))) → ABC^D-+EAC/-*

14.Define Queue

Queue is an ordered collection of homogeneous data elements, in which the element insertion and deletion takes place at two ends called front and rear. The elements are ordered in linear fashion and inserted at REAR end and deleted FRONT end. This exhibits FIFO (First In First Out) property.

15.Applications of Queue

Applications of queue as a data structure are more common.

a) Within a computer system there may be queues of tasks waiting for the line printer, or for access to disk storage, or in a time-sharing system for use of the CPU.
b) Within a single program, there may be multiple requests to be kept in a queue, or one task may create other tasks, which must be done in turn by keeping them in a queue.

16.What is the need of Circular Queue?

- Queue implemented using an array suffers from one limitation i.e. there is a possibility that the queue is reported as full *(since rear has reached the end of the array)*, even though in actuality there might be empty slots at the beginning of the queue. To overcome this limitation ***circular queue*** is needed.
- Now the queue would be reported as full only when all the slots in the array stand occupied.

17.What is deque?

- The word deque is a short form of ***double-ended queue*** and defines a data structure in which items can be added or deleted at either the front or rear end, but no changes can be made elsewhere in the list.
- Thus a deque is a generalization of both a stack and a queue.

18.Types of Linked Lists:

a) Linear Singly Linked List
b) Circular Linked List
c) Two-way or doubly linked lists
d) Circular doubly linked lists

19.What are the Applications of linked list?

a) Implementation of Stack
b) Implementation of Queue
c) Implementation of Tree
d) Implementation of Graph

20.Applications of Stack

a) It is very useful to evaluate arithmetic expressions. (Postfix Expressions)
b) Infix to Postfix Transformation

c) It is useful during the execution of recursive programs
d) A Stack is useful for designing the compiler in operating system to store local variables inside a function block.
e) A stack can be used in function calls including recursion.
f) Reversing Data
g) Reverse a List
h) Convert Decimal to Binary
i) Parsing – It is a logic that breaks into independent pieces for further processing
j) Backtracking

UNIT II: TREE STRUCTURES

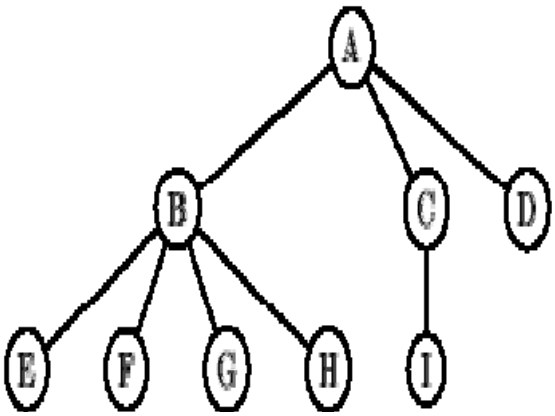
Need for non-linear structures – Tree ADT – tree traversals – left child right sibling data structures for general trees – Binary Tree ADT – expression trees – applications of trees – binary search tree ADT

Unit II

TREES

2.1 The ADT tree

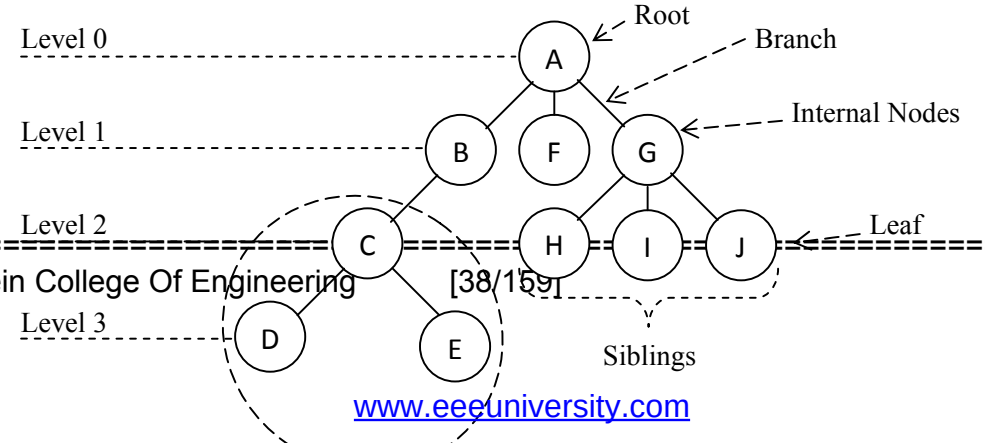
A **tree** is a finite set of elements or **nodes**. If the set is non-empty, one of the nodes is distinguished as the **root** node, while the remaining (possibly empty) set of nodes are grouped into subsets, each of which is itself a tree. This hierarchical relationship is described by referring to each such subtree as a **child** of the root, while the root is referred to as the **parent** of each subtree. If a tree consists of a single node, that node is called a **leaf** node.



A simple tree.

2.1.1 Basic Tree Concepts

- A tree consists of a finite set of elements, called ***'nodes'***, and a finite set of directed lines, called ***'branches'***, that connect the nodes.
- The number of branches associated with a node is the ***degree of the node***.
  - When the branch is directed toward a node, it is an ***indegree branch***; when the branch is directed away from the node, it is an ***outdegree branch***.
  - *The sum of indegree and outdegree branches is the degree of the node.*
  - The indegree of the root is by definition is zero.
- A ***leaf*** is any node with an outdegree of zero.
- A node that is not a root or a leaf is known as an ***internal node***.
- A node is a ***parent*** if it has ***successor*** nodes – that is, if it has an outdegree greater than zero.
- Conversely, a node with a **predecessor** is a ***child***. A child node has an indegree of one.
- Two or more nodes with the same parent are ***siblings***.

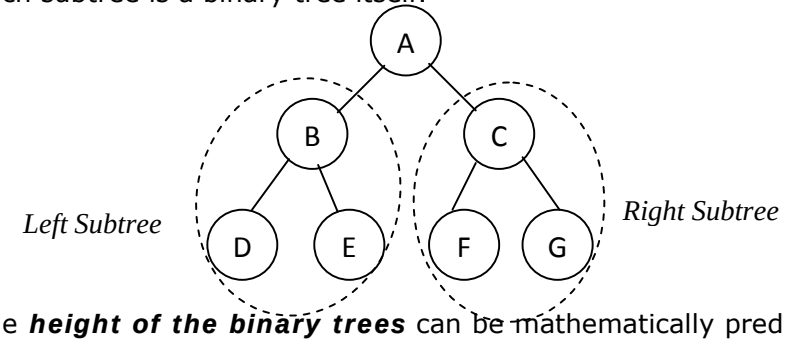


Parents	:	A, B, C, G
Children	:	B, F, G, C, H, I, J, D, E
Siblings	:	{ B, F, G }, { D, E }, { H, I, J }
Leaves	:	F, D, E, H, I, J
Length	:	4

- A ***path*** is a sequence of nodes in which each node is adjacent to the next one.
- Every node in the tree can be reached by following a ***unique path*** starting from the root.
- The ***level*** of a node is its distance from the root. Because the root has a zero distance from itself, the root is at level 0. The children of the root are at the level 1.
- The ***height*** or ***length*** of the tree is the level of the leaf in the longest path from the root plus 1. By definition, the height of an empty tree is -1.
- A tree may be divided into ***subtrees***. A ***subtree*** is any connected structure below the root.
- The first node in a subtree is known as the ***root of the subtree*** and is used to name the subtree.

Binary Trees

- A ***binary tree*** is a tree in which no node can have more than two subtrees.
- These subtrees are designated as the ***left subtree*** and ***right subtree***.
- Each subtree is a binary tree itself.



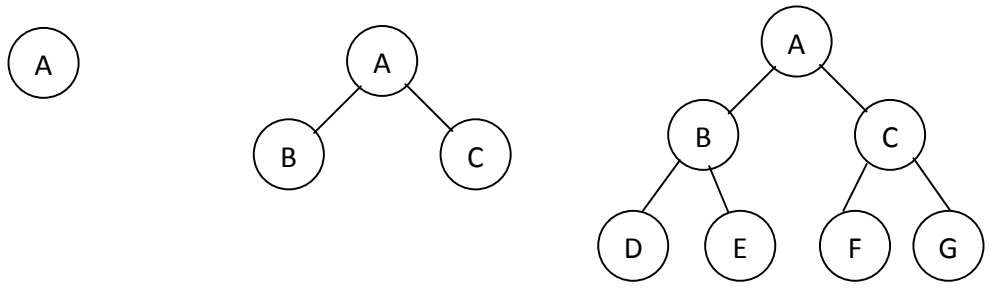
- The ***height of the binary trees*** can be mathematically predicted. The maximum height of the binary tree which has N nodes, $\mathbf{H_{max} = N}$
- A tree with a maximum height is rare. It occurs when the entire tree is built in one direction. The ***minimum height of the tree***, H_{min} is determined by,

$$\mathbf{Hmin = \lfloor Log_2 N \rfloor + 1}$$

- Given a height of the binary tree, H, the minimum and maximum number of nodes in the tree are given as, $\mathbf{N_{min} = H}$ **and,**

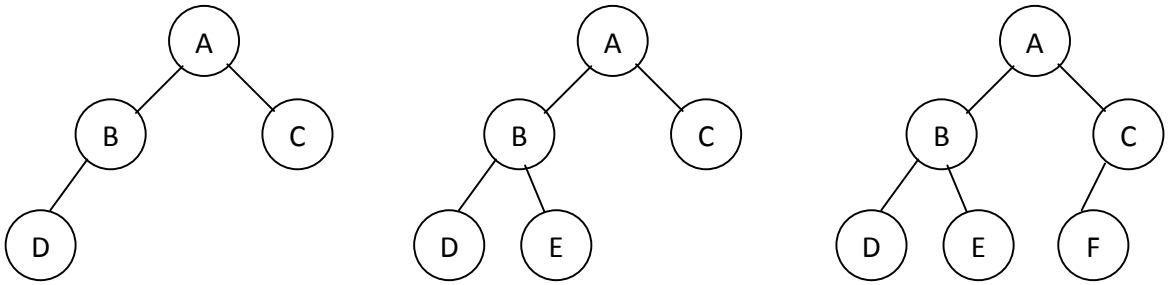
$$\mathbf{N_{max} = 2^H - 1}$$

- If the height of the tree is less, then it is easier to locate any desired node in the tree.
- To determine whether tree is balanced, the ***balance factor*** should be calculated.
- If H_L represents the height of the left subtree and H_R represents the height of the right subtree then ***Balance factor***, $\mathbf{B = H_L - H_R}$
- A tree is balanced if its balance factor is ***0*** and its subtrees are also balanced.
- A binary tree is balanced if the *height of its subtrees differs by no more than one and its subtrees are also balanced*.
- A ***complete tree*** has the maximum number of entries for its height.
- The maximum number is reached when the least level is full. *The maximum number is reached when the last level is full*.



Complete Trees (at levels 0, 1, and 2)

- A tree is considered ***nearly complete*** if it has the minimum height for its nodes and all nodes in the last level are found on the left.



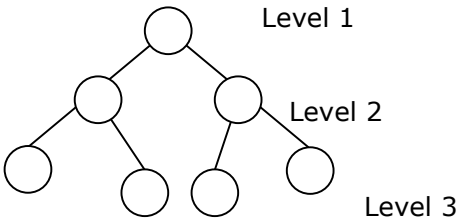
Nearly Complete Trees (at level 2)

2.1.2 Binary Tree Representation

A **binary tree** is a tree which is either empty, or one in which every node:

- has no children; or
- has just a left child; or
- has just a right child; or
- has both a left and a right child.

A Complete binary tree of depth K is a binary tree of depth K having 2k – 1 nodes.



A very simple representation for such binary tree results from sequentially numbering the nodes, starting with nodes on level 1 then those on level 2 and so on. Nodes on any level are numbered from left to right as shown in the above picture. This numbering scheme gives us the definition of a complete binary tree. A binary tree with n nodes and of depth K is complete if its nodes correspond to the nodes which are numbered one to n in the full binary tree of depth K.

Array Representation:

Each node contains **info, left, right** and **father** fields. The left, right and father fields of a node point to the node's left son, right son and father respectively.

Using the array implementation, we may declare

```
#define NUMNODES 100
struct nodetype
        {
        int info;
        int left;
        int right;
        int father;
        };
struct nodetype node[NUMNODES];
```

This representation is called linked array representation.

Under this representation,

info(p) would be implemented by reference **node[p].info,**

left(p) would be implemented by reference **node[p].left,**

right(p) would be implemented by reference **node[p].right,**

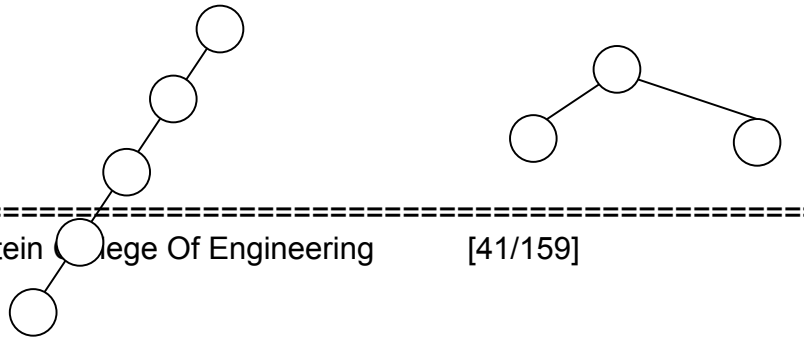
father(p) would be implemented by reference **node[p].father** respectively.

The operations,

isleft(p) can be implemented in terms of the operation **left(p)**

isright(p) can be implemented in terms of the operation **right(p)**

Example: -



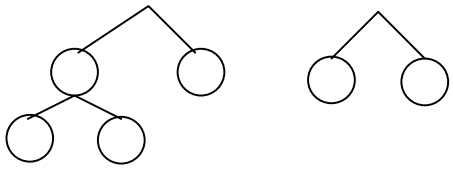
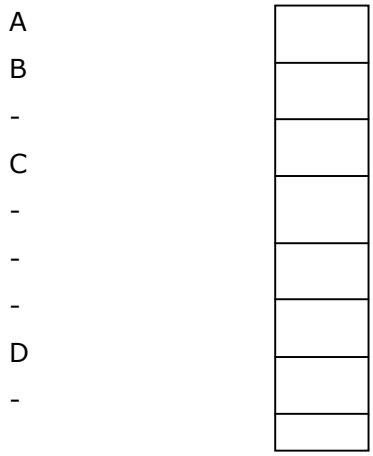


Fig (a) Fig (b)

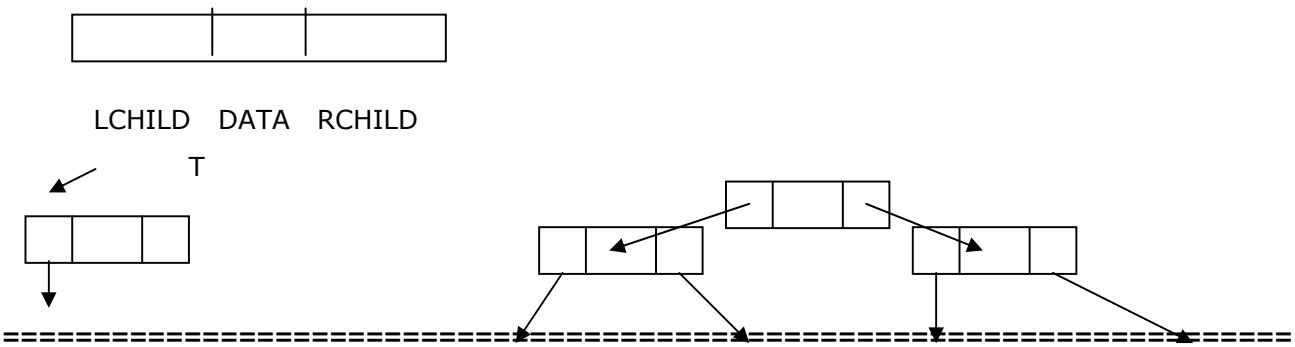
The above trees can be represented in memory sequentially as follows



The above representation appears to be good for complete binary trees and wasteful for many other binary trees. In addition, the insertion or deletion of nodes from the middle of a tree requires the insertion of many nodes to reflect the change in level number of these nodes.

Linked Representation: -

The problems of sequential representation can be easily overcome through the use of a linked representation. Each node will have three fields LCHILD, DATA and RCHILD as represented below



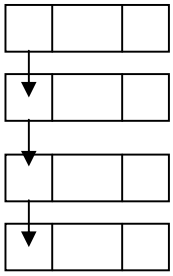


Fig (a)

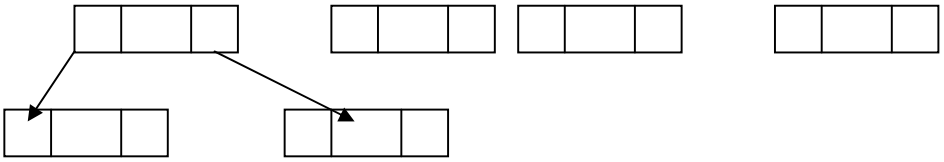


Fig (b)

In most applications it is adequate. But this structure make it difficult to determine the parent of a node since this leads only to the forward movement of the links.

Using the linked implementation, we may declare,

```
stuct nodetype
        {
        int info;
        struct nodetype *left;
        struct nodetype *right;
        struct nodetype *father;
        };
typedef struct nodetype *NODEPTR;
```

This representation is called **dynamic node representation.** Under this representation,

info(p) would be implemented by reference **p→info,**

left(p) would be implemented by reference **p→left,**

right(p) would be implemented by reference **p→right**,

father(p) would be implemented by reference **p→father**.

Primitive Operation On Binary Trees

(1) maketree() function

Which allocates a node and sets it as the root of a single-node binary tree, may be written as follows;

```
NODEPTR maketree(x)
int x;
{
NODEPTR p;
p = getnode();              / *  getnode() function  get a  available node  */
p→info = x;
p→left = NULL;
p→right=NULL;
return(p);
}
```

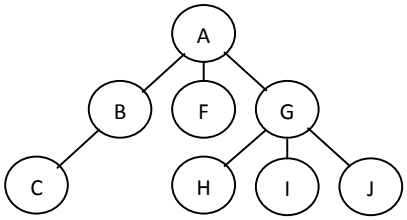
(2) setleft(p,x) function Which sets a node with contents x as the left son of node(p)

```
setleft(p,x)
NODEPTR p;
int x;
{
if(p = = NULL)
        printf("insertion not made");
else if ( p→left ! = NULL)
        printf("invalid insertion ");
else
        p→left = maketree (x);
}
```

Conversion of a General Tree to Binary Tree

General Tree:

- A General Tree is a tree in which each node can have an unlimited out degree.
- Each node may have as many children as is necessary to satisfy its requirements. *Example: Directory Structure*



- It is considered easy to represent binary trees in programs than it is to represent general trees. So, the general trees can be represented in binary tree format.

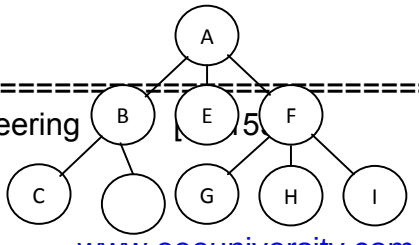
Changing general tree to Binary tree:

- The binary tree format can be adopted by changing the meaning of the left and right pointers. There are two relationships in binary tree,
  - Parent to child
  - Sibling to sibling

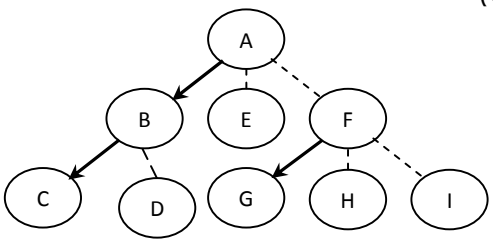
  Using these relationships, the general tree can be implemented as binary tree.

Algorithm

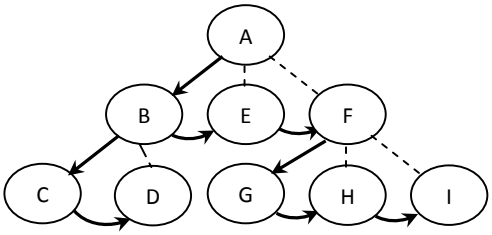
1. Identify the branch from the parent to its first or leftmost child. These branches from each parent become left pointers in the binary tree
2. Connect siblings, starting with the leftmost child, using a branch for each sibling to its right sibling.
3. Remove all unconnected branches from the parent to its children



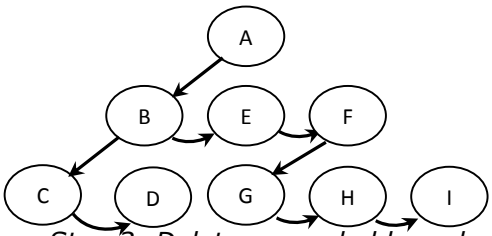
(a) General Tree



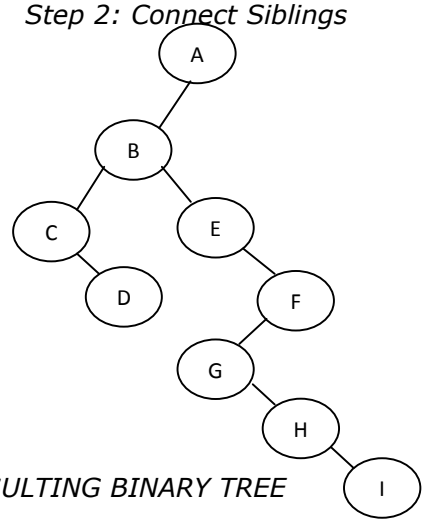
Step 1: Identify all leftmost children



Step 2: Connect Siblings



Step 3: Delete unneeded branches



(B) THE RESULTING BINARY TREE

2.1.3 Binary Tree Traversals

- A binary tree traversal requires that each node of the tree be processed once and only once in a predetermined sequence.
- The two general approaches to the traversal sequence are,
  - ***Depth first traversal***
  - ***Breadth first traversal***
- In depth first traversal, the processing proceeds along a path from the root through one child to the most distant descendent of that first child before processing a second child. *In other words, in the depth first traversal, all the descendants of a child are processed before going to the next child*.
- In a breadth-first traversal, the processing proceeds horizontally form the root to all its children, then to its children's children, and so forth until all nodes have been processed. *In other words, in breadth traversal, each level is completely processed before the next level is started*.

a) Depth-First Traversal

There are basically three ways of binary tree traversals. They are :

1. **Pre Order Traversal**
2. **In Order Traversal**
3. **Post Order Traversal**

In C, each node is defined as a structure of the following form:

```
struct node
        {
        int  info;
        struct node *lchild;
        struct node *rchild;
        }
typedef struct node NODE;
```

Binary Tree Traversals (Recursive procedure)

1. Inorder Traversal

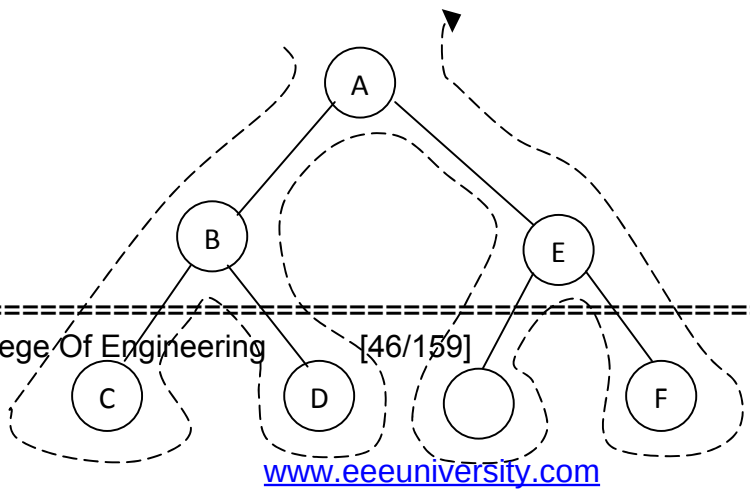
Steps : 1. Traverse left subtree in inorder
2. Process root node
3. Traverse right subtree in inorder

Algorithm

```
Algorithm inoder traversal (Bin-Tree T)

Begin
If ( not empty (T) ) then

Begin
        Inorder_traversal ( left subtree ( T ) )
        Print ( info ( T ) )   /  * process node */
        Inorder_traversal ( right subtree ( T ) )
End
End
```

C Coding

```
void inorder_traversal ( NODE * T)
  {
if( T ! = NULL)
        {
        inorder_traversal(T->lchild);

        printf("%d \t ", T->info);
        inorder_traversal(T->rchild);
                }
}
```



The Output is : C → B → D →A →E → F

2. Preorder Traversal

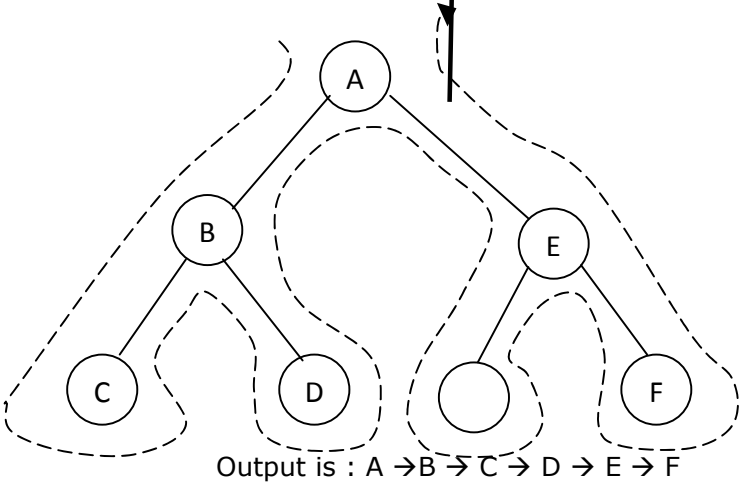
Steps : 1. Process root node
2. Traverse left subtree in preorder
3. Traverse right subtree in preorder

Algorithm

```
Algorithm preorder traversal (Bin-Tree T)

Begin
If ( not empty (T) ) then
Begin
        Print ( info ( T ) ) /  * process node * /
        Preoder traversal (left subtree ( T ) )
        Inorder traversal ( right subtree ( T ) )
End
End
```

C function

```
void preorder_traversal ( NODE * T)
  {
if( T ! = NULL)
        {
        printf("%d \t", T->info);
        preorder_traversal(T->lchild);
        preorder_traversal(T->rchild);
        }
         }
```



Output is : A →B → C → D → E → F

3. Postorder Traversal

Steps : 1. Traverse left subtree in postorder
2. Traverse right subtree in postorder
3. process root node

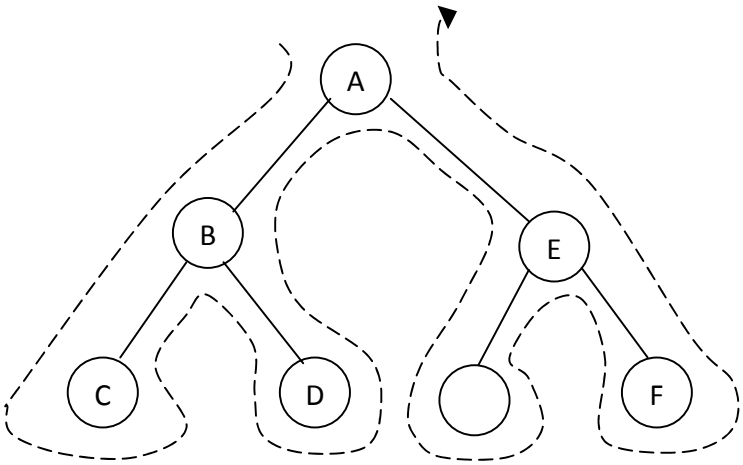
Algorithm

C function

Postorder Traversal

```
Algorithm postorder traversal (Bin-Tree T)

Begin
If ( not empty (T) ) then
Begin
        Postorder_traversal ( left subtree ( T ) )
        Postorder_traversal ( right subtree( T))
        Print ( Info ( T ) ) / * process node */
End
End
```

```
void postorder_traversal ( NODE * T)
  {
   if( T ! = NULL)
        {
        postorder_traversal(T->lchild);
        postorder_traversal(T->rchile);
        printf("%d \t", T->info);
        }
   }
```



The Output is : C → D → B → F → E → A

Non – Recursive algorithm: Inorder_Traversal

```
#define MAXSTACK 100
inorder_traversal (tree)
NODEPTR tree;
        {
        struct stack
                {
                int top;
                NODEPTR item[MAXSTACK];
                }s;
        NODEPTR  p;
s.top = -1;
p = tree;
do
        {/*  travel down left branches as far as possible, saving
```

```
        pointers to nodes passed */
    while(p!=NULL)
        {
        push(s,p);
        p = p → left;
        }
        /*  check if finished */
    if ( !empty (s) )
        {                           /  * at this point the left subtree is empty */
        p=pop(s);
        printf("%d \n", p→info);        /* visit the root */
        p =p→right;                     /*  traverse right subtree */
        }
    }while( !empty (s) || p! = NULL );
}
```

Non – Recursive algorithm: Pre**order_Traversal**

```
#define MAXSTACK 100
preorder_traversal (tree)
{
NODEPTR tree;
    {
    struct stack
        {
        int top;
        NODEPTR item[MAXSTACK];
        }s;
    NODEPTR  p;
s.top = -1;
p = tree;
do
    {
    /*  travel down left branches as far as possible, saving
        pointers to nodes passed */
    if(p!=NULL)
        {
```

```
            printf("%d \n", p→info);                    /* visit the root */
            if(p→right!=NULL)
                  push(s,p→right);      /* push the right subtree on to the stack */
            p=p→left;
            }
      else
            p=pop(s);
      }while( ! empty(s) || p! = NULL )
}
```

2.2 Binary Search Tree

Binary tree that all elements in the left subtree of a node n are less than the contents of n, and all elements in the right subtree of n are greater than or equal to the contents of n.

Uses : used in sorting and searching

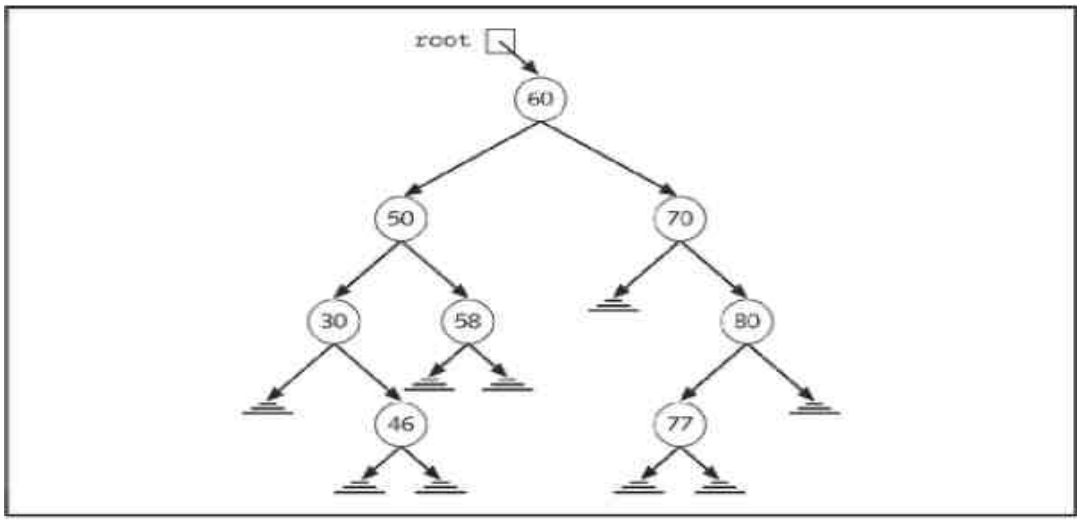


Figure Binary search tree

Applications of Binary Trees (find the duplication element from the list)

Binary tree is useful data structure when two-way decisions must be made at each point in a process. For example find the all duplicates in a list of numbers. One way doing this is each number compare with it's precede it. However, it involves a large number of comparisons.

The number of comparisons can be reduced by using a binary tree.

Step 1: from root, each successive number in the list is then compared to the number in the root.

Step 2: If it matches, we have a duplicate.

Step 3: If it is smaller, we examine the left subtree.

Step 4: If it is larger, we examine the right subtree.

Step 5: If the subtree is empty, the number is not a duplicate and is placed into a new node at that passion in the tree.

Step 6: If the subtree is nonempty, we compare the number of the contents of root of the subtree and entire process is repeated till all node completed.

```
/*  read the first number and insert it into a sinlge-node binary tree */
        scanf("%d",&number);
        tree = maketree (number);
        while(there are numbers left in the input)
                {
                scanf("%d", &number);
                p = q = tree;
                while ( number ! = info(p) && q ! = NULL)
                        {
                        p = q;
                        if ( number < info (p) )
                                q = left (p);
                        else
                                q = right (p);
                        }
                if(number = = info(p) )
                        printf(" %d %s ", number, "is a duplicate");
                else if ( number < info(p) )
                        setleft( p, number );
                else
                        setright(p, number);
                }
```

(2) Application of Bianry Tree - (Sort the number in Ascending Order)

If a binary search tree is traversed in inorder(left,root,right) and the contents of each node are printed as the node is visited, the numbers are printed in ascending order.

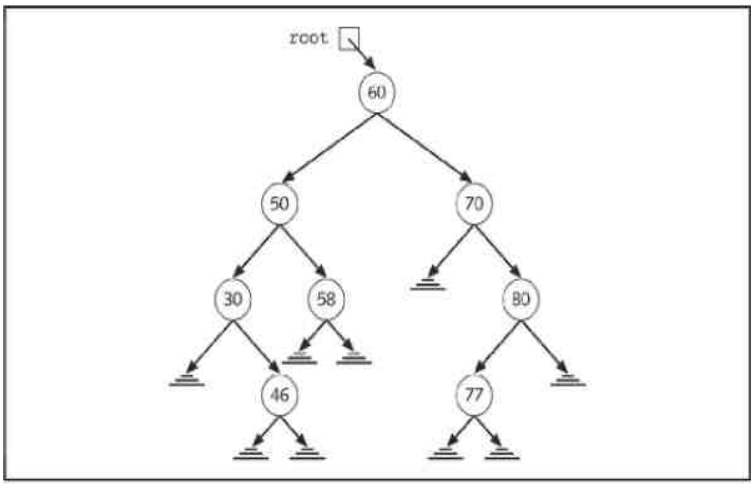


Figure Binary search tree

For convince the above binary search tree if it is traversed inorder manner the result order is,

30, 46, 50, 58, 60, 70, 77 and 80 is ascending order sequence.

(3) Application Binary Tree – (Expression Tree)

Representing an expression containing operands and binary operators by a strictly binary tree. The root of the strictly binary tree contains an operator that is to be applied to the results of evaluating the expressions represented by the left and right subtrees. A node representing an operator is a nonleaf, whereas a node representing an operand in a leaf.

BINARY SEARCH TREE OPERATIONS:

The basic operation on a binary search tree(BST) include, creating a BST, inserting an element into BST, deleting an element from BST, searching an element and printing element of BST in ascending order.

The ADT specification of a BST:

ADT BST

{

Create BST()	:	Create an empty BST;
Insert(elt)	:	Insert elt into the BST;
Search(elt,x)	:	Search for the presence of element elt and Set x=elt, return true if elt is found, else return false.
FindMin()	:	Find minimum element;
FindMax()	:	Find maximum element;
Ordered Output()	:	Output elements of BST in ascending order;

```
Delete(elt,x)          :  Delete elt and set x = elt;
}
```

Inserting an element into Binary Search Tree

```
Algorithm InsertBST(int elt, NODE *T)
[ elt is the element to be inserted and T is the pointer to the root of the tree]
  If (T= = NULL) then
        Create a one-node tree and return
Else if (elt<key) then
        InsertBST(elt, T→lchild)
Else if(elt>key) then
        InsertBST(elt, T→rchild)
Else
        " element is already exist
return T
End
```

C coding to Insert element into a BST

```
struct node
        {
        int info;
        struct node *lchild;
        struct node *rchild;
        };
typedef stuct node NODE;
NODE *InsertBST(int elt, NODE *T)
        {
        if(T = = NULL)
                {
                T = (NODE *)malloc(sizeof(NODE));
                if ( T = = NULL)
                        printf ( " No memory error");
                else
                        {
                        t→info = elt;
                        t→lchild = NULL;
                        t→rchild = NULL;
                        }
                }
```

```
        else if ( elt < T→info)
                t→lchild = InsertBST(elt, t→lchild);
        else if ( elt > T→info)
                t→rchild = InsertBST( elt, t→rchild);
        return T;
        }
```

Searching an element in BST

Searching an element in BST is similar to insertion operation, but they only return the pointer to the node that contains the key value or if element is not, a NULL is return;

Searching start from the root of the tree;

If the search key value is less than that in root, then the search is left subtree;

If the search key value is greater than that in root, then the search is right subtree;

This searching should continue till the node with the search key value or null pointer(end of the branch) is reached.

In case null pointer(null left/righ chile) is reached, it is an indication of the absence of the node.

```
Algorithm SearchBST(int elt, NODE *T)
[ elt is the element to be inserted and T is the pointer to the root of the tree]
1. If (T= = NULL) then
        Return NULL
2. If (elt<key) then                    /* elt is less than the key in root */
        return SearchBST(elt, T→lchild)
Else if(elt>key) then                   /* elt is greater than the key in root */
        return SerachBST(elt, T→rchild)
Else
        return T
End
NODE * SearchBST(int elt, NODE *T)
        {
        if(T = = NULL)
                return NULL;

         if ( elt < T→info)
                return SearchBST(elt, t→lchild);
        else if ( elt > T→info)
                return SearchBST( elt, t→rchild);
```

```
        else
                return T;
        }
```

Finding Minimum Element in a BST

Minimum element lies as the left most node in the left most branch starting from the root. To reach the node with minimum value, we need to traverse the tree from root along the left branch till we get a node with a null / empty left subtree.

Algorithm FindMin(NODE * T)

1. If Tree is null then
   return NULL;
2. if lchild(Tree) is null then
   return tree
   else
   return FindMin(T→lchild)
3. End

```
NODE * FindMin( NODE *T )
        {
        if(T = = NULL)
                return NULL;

         if ( T→lchild = = NULL)
                return tree;
        else
                return FindMin(Tree→lchild);
        }
```

Finding Maximum Element in a BST

Maximum element lies as the right most node in the right most branch starting from the root. To reach the node with maximum value, we need to traverse the tree from root along the right branch till we get a node with a null / empty right subtree.

Algorithm FindMax(NODE * T)

1. If Tree is null then
   return NULL;
2. if rchild(Tree) is null then
   return tree
   else
   return FindMax(T→rchild)

3. End

```
NODE * FindMax( NODE *T )
        {
        if(T = = NULL)
                return NULL;

         if ( T→rchild = = NULL)
                return tree;
        else
                return FindMax(Tree→rchild);
        }
```

DELETING AN ELEMENT IN A BST

The node to be deleted can fall into any one of the following categories;

1. Node may not have any children (ie, it is a leaf node)
2. Node may have only on child (either left / right child)
3. Node may have two children (both left and right)

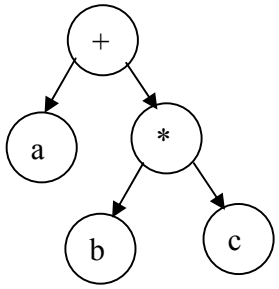
```
Algorithm DeleteBST(int elt, NODE * T)
1. If Tree is null then
        print "Element is not found"
2. If elt is less than info(Tree) then
        locate element in left subtree and delete it
    else if elt is greater than info(Tree) then
        locate element in right subtree and delete it
    else if (both left and right child are not NULL) then    /* node with two children */
        begin
                Locate minimum element in the right subtree
                Replace elt by this value
                Delete min element in right subtree and move the remaining tree as its
                right child
        end
   else
        if leftsubtree is Null then
                /*  has only right subtree or both subtree Null */
                replace node by its rchild
        else
```

```
                if right subtree is Null then
                        replace node by its left child\
        end
        free memory allocated to min node
        end
return Tree
End

NODE *DeleteBST(int elt, NODE * T)
{
NODE * minElt;
if(T = = NULL)
        printf(" element not found \n");
else if ( elt < T→info)
        T→lchild = DeleteBST(elt, T→lchild);
else if ( elt > T→info)
        T→rchild = DeleteBST(elt, T→rchild);
else
        if(T→lchild && T→ rchild)
                {
                minElt = FindMin(T→rchild);
                T→info = minElt→info;
                T→rchild = DeleteBST(T→info, T→rchild);
                }
        else
                {
                minElt = Tree;
                if (T→lchild = = NULL)
                        T = T→rchild;
                else if ( T→rchild = = NULL)
                        T = T→lchild;
                Free (minElt);
                }
return T;
}
```

2.3 Expression tree:

Expression Tree is a binary tree in which the leaf nodes are operands and the interior nodes are operators. Like binary tree, expression tree can also be travesed by inorder, preorder and postorder traversa

For ex. Expression tree for a+b*c is as follows:



2.3.1 Algorithm for Constructing an Expression Tree with an example:

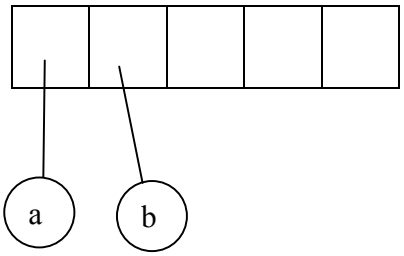
Let us consider postfix expression given as an input for constructing an expression tree by performing the following steps:

1. Read one symbol at a time from the postfix expression.
2. Check whether the symbol is an operand or operator.
   (a) If the symbol is an operand, create a one - node tree and push a pointer on to the stack.
   (b) If the symbol is an operator pop two pointers from the stack namely T_1 and T_2 and form a new tree with root as the operator and T_2 as a left child and T_1 as a right child. A pointer to this new tree is then pushed onto the stack.

Consider the input, ab+cde+**

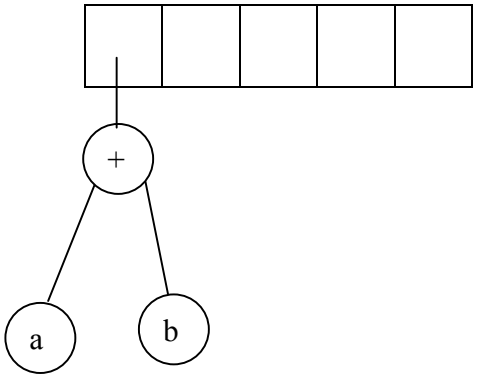
Step 1:

The first two symbols are operands, so create one-node tree and push pointers of them onto a stack.



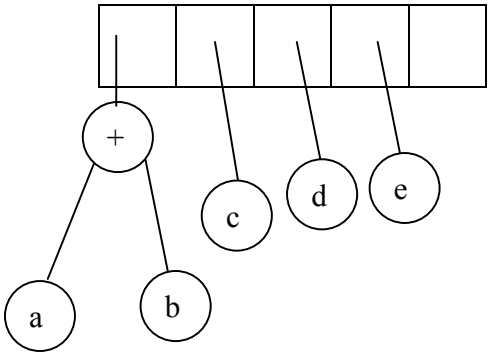
Step 2:

Next symbol is read, i.e '+' operator. Now the symbol is operator, so two previous pointers to trees in the stack are popped, and a new tree is formed and its pointer is pushed on to the stack. The popped pointers are placed as left child and right child in the new tree, as a result two subtrees are merged.



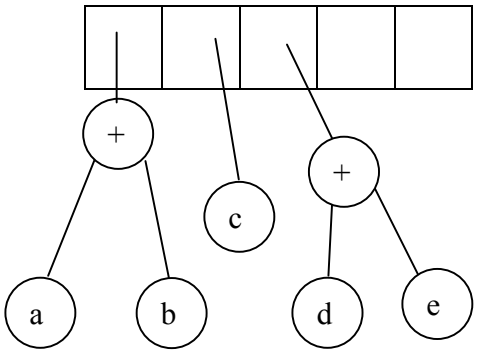
Step 3:

Next symbols are operands i.e., c,d,and e, they are read. For each operand create one-node tree and push pointers of the corresponding tree onto the stack.



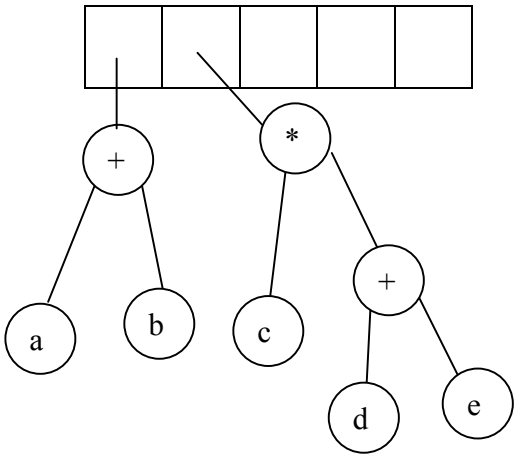
Step 4:

Next symbol is read, i.e '+' operator. Now the symbol is operator, so two previous pointers to trees in the stack are popped, and a new tree is formed and its pointer is pushed on to the stack. The popped pointers are placed as left child and right child in the new tree, as a result two subtrees are merged.



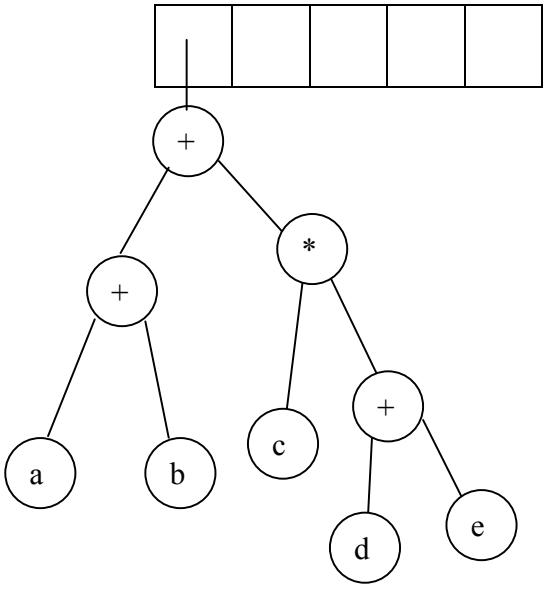
Step 5:

Next symbol is read, i.e '*' operator. The symbol is operator, so two previous pointers to trees in the stack are popped, and a new tree is formed and its pointer is pushed on to the stack. The popped pointers are placed as left child and right child in the new tree, as a result two subtrees are merged.



Step 6:

Final symbol is read, i.e '*' operator. The symbol is operator, so two previous pointers to trees in the stack are popped, and a new tree is formed and its pointer is pushed on to the stack. The popped pointers are placed as left child and right child in the new tree, as a result two subtrees are merged. So a pointer to the final tree is left on the stack.



Atlast we made an expression tree. In this tree we can make in-order , pre-order and post- order traversal like a binary tree.

2.3.2 Algorithm & Implementation of Expression Tree

Algorithm:

1. Start the program.
2. Create two stacks.
    a. Operator stack- for operators.
    b. Operand Stack- for tree node built so far.
3. From the infix Expression, Process one token at a time.
4. Each time we see an operand, we create a new BinaryNode for the operand, and push a pointer to the operand stack.
5. Each time we see an operator, compare the precedence of this input operator with the precedence of operator in top of operator stack. If the Precedence is high, push the input operator on to the stack. Otherwise pop the operators from the until the stack operators precedence is low.
6. When we pop an operator from the operator stack, we create a new BinaryNode for it and we pop two operands from the operand stack, we pop the first one and make it the right child of a operator node and pop the second one and make it the left child. Then we push the operator node pointer onto the operand stack.
7. At the end, the tree node on the top of the operand stack is the root of the expression tree for the given infix expression.
8. For in order traversal

a. Visit Left sub tree
b. Visit Root node
c. Visit Right sub tree.

9. For preorder traversal.
   a. Visit Root node
   b. Visit Left sub tree
   c. Visit Right sub tree
10. For post order traversal
   a. Visit Right sub tree
   b. Visit Left sub tree
   c. Visit Root node.
11. Stop the program.

2.3.3 Implementation of Expression tree and make in-order, pre-order and post-order traversal

```
#include<stdio.h>
struct tree
{
        char info;
        struct tree *rchild;
        struct tree *lchild;
};

int prec(char data);
typedef struct tree * node;

char pop_op();
node pop_num();
void push_op(char item);

node create()
{
        return((node)malloc(sizeof(node)));
}
node num[20],root=NULL;
char op[20],oprt,ev[20];
int nt=-1,ot=-1,et=-1;

main()
{
        node newnode,item,temp;
        char str[50];
        int i,k,p,s,flag=0;
        printf("ENTER THE EXPRESSION ");
        scanf("%s",str);
        printf("\n%s",str);
        for(i=0;str[i]!='\0';i++)
        {
                if(isalnum(str[i]))
                {
                        newnode=create();
                        newnode->info=str[i];
                        newnode->lchild=NULL;
                        newnode->rchild=NULL;
                        item=newnode;
                        push_num(item);
                }
```

```
                else
                {
                        if(ot!=-1)
                        p=prec(op[ot]);
                        else
                        p=0;
                        k=prec(str[i]);
                        if(k==5)
                        {
                                while(k!=1)
                                {
                                        oprt=pop_op();
                                        newnode=create();
                                        newnode->info=oprt;
                                        newnode->rchild=pop_num();
                                        newnode->lchild=pop_num();
                                        // if(root==NULL)
                                        root=newnode;
                        // else if((newnode->rchild==root)||(newnode->lchild==root))
                        // root=newnode;
                                        push_num(root);
                                        k=prec(op[ot]);
                                }
                                oprt=pop_op();
                        }
                else if(k==1)
                push_op(str[i]);
                else
                {
                        if(k>p)
                        push_op(str[i]);
                        else
                        {
                                if(k<=p)
                                {
                                        oprt=pop_op();
                                        newnode=create();
                                        newnode->rchild=pop_num();
                                        newnode->lchild=pop_num();
                                        if(root==NULL)
                                        root=newnode;
                                else if((newnode->rchild==root)||(newnode->lchild==root))
                                        root=newnode;
                                        push_num(newnode);
                                        push_op(str[i]);
                                        // k=prec(op[ot]);
                                }
                        }
                }
        }
}
        printf("\nThe prefix expression is\n ");
        preorder(root);
        printf("\nThe infix exp is\n ");
        inorder(root);
        printf("\nThe postfix expression is\n ");
        postorder(root);
        evaluate();
}

void push_op(char item)
{
```

```
        op[++ot]=item;
}

push_num(node item)
{
        num[++nt]=item;
}

char pop_op()
{
        if(ot!=-1)
        return(op[ot--]);
        else
        return(0);
}

node pop_num()
{
        if(nt!=-1)
        return(num[nt--]);
        else
        return(NULL);
}

int prec(char data)
{
        switch(data)
        {
                case '(':return(1);
                        break;
                case '+':
                case '-':return(2);
                        break;
                case '*':
                case '/':return(3);
                        break;
                case '^':return(4);
                        break;
                case ')':return(5);
                        break;
        }
}

inorder(node temp)
{
        if(temp!=NULL)
        {
                inorder(temp->lchild);
                printf("%c ",temp->info);
                inorder(temp->rchild);
        }
}

preorder(node temp)
{
        if(temp!=NULL)
        {
                printf("%c ",temp->info);
                preorder(temp->lchild);
                preorder(temp->rchild);
        }
}
```

```
postorder(node temp)
{
        if(temp!=NULL)
        {
                postorder(temp->lchild);
                postorder(temp->rchild);
                printf("%c ",temp->info);
                ev[++et]=temp->info;
        }
}

evaluate()
{
        int i,j=-1,a,b,ch[20];
        for(i=0;ev[i]!='\0';i++)
        {
                if(isalnum(ev[i]))
                ch[++j]=ev[i]-48;
                else
                {
                        b=ch[j];
                        a=ch[j-1];
                        switch(ev[i])
                        {
                                case '+':ch[--j]=a+b;
                                        break;
                                case '-':ch[--j]=a-b;
                                        break;
                                case '*':ch[--j]=a*b;
                                        break;
                                case '/':ch[--j]=a/b;
                                        break;
                        }
                }
        }
        printf("\nValue = %d",ch[0]);
}
```

UNIT III : BALANCED SEARCH TREES AND INDEXING

AVL trees – Binary Heaps – B-Tree – Hashing – Separate chaining – open addressing –Linear probing

UNIT III
BALANCED SEARCH TREES&INDEXING

Height- Balanced Tree: An empty tree is height-balanced if T is a non empty binary tree with T_L and T_R as its left and eight sub trees respectively, then T is a height –balanced if

1. T_L and T_R are height-balanced
2. $| h_L - h_R | \leq 1$ where h_L and h_R are the heights of T_L and T_R respectively.

The definition of a height balanced binary tree requires that every sub tree also be height balanced. The balance factor of the nodes decides if the tree is balanced or not.

Balance Factor: The balance factor BF(T) of a node T is a binary tree is defined to be

$$B.F = h_L - h_R$$

where h_L and h_R respectively are the heights of the left and right sub trees of T.

For any node T is an AVL tree BF(T) = -1, 0, or 1. If it is not in among these 3 values then the tree has to be rebalanced by performing the appropriate rotation.

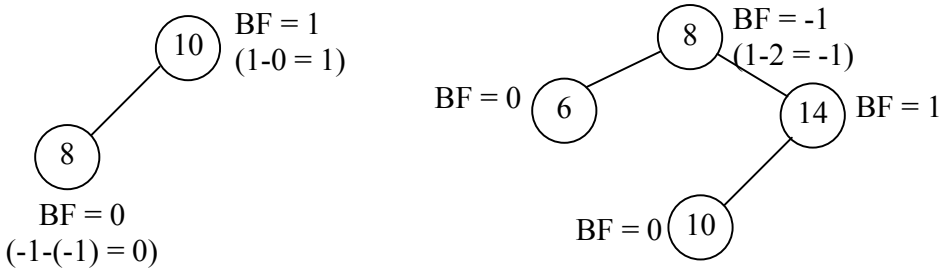
3.1 AVL TREE:(Balanced Search Tree)

Adelson – Velskii –and Landis introduced a binary tree structure that is balanced with respect to heights of sub trees. The order of retrieval, insertion and deletion is only O (log n). This tree structure is called AVL tree.

An AVL tree is a special type of binary tree that is always "partially" balanced. The criteria that is used to determine the "level" of "balanced-tree" is the difference between the heights of sub trees of a root in the tree. The "height of tree is the "number levels" in the tree. To be more formal, the height of a tree is defined as follows:

1. The height of a tree with no elements is 0
2. The height of a tree with 1 element is 1
3. The height of tree with > 1 element is equal to 1 + height of its tallest sub tree

An AVL tree is a binary tree in which the difference between the height of the right and left sub trees or the root node is never more than one.



3.1.1 Insertion of a node:

We can insert a new node into an AVL tree by first using binary tree insertion algorithm, comparing the key of the new node with that in the root, and inserting the new node into the left or right sub tree as appropriate.

It often turns out that the new node can be inserted without changing the height of a sub tree, in which case neither the height not the balance of the root will be changed. Even when the height of a sub tree does increase, it may be the shorter subtree that has grown, so only the balance factor of the root will change.

3.1.2 Rotations:

When a new node has been inserted into the taller subtree of the root and its height has increased, so that now one subtree has height 2 more than the other, and the tree no longer satisfies the AVL requirements. We must now rebuild part of the tree to restore its balance.

The rebalancing can be carried out using four different kinds of rotations: LL, RR, LR & RL. LL and RR are symmetric as are LR and RL. There rotations are characterized by the nearest ancestor, A, of the inserted node, Y, whose balance factor become ± 2. The following rotations type is obtained:

LL: New node Y is inserted in the Left Subtree of the Left Subtree of A → Single Rotation with left.

LR: Y is inserted in the right subtree of the Left Subtree of A → Double Rotation with Left

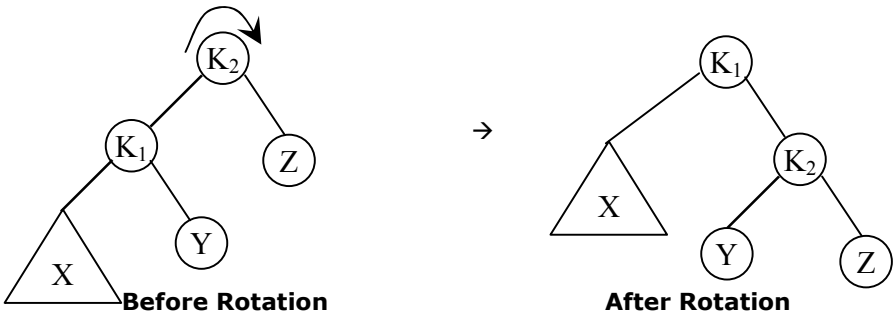
RR: Y is inserted in the right subtree of the right subtree of A → Single Rotation with Right

RL: Y is inserted in the left subtree of the right subtree of A → Double Rotation with Right

Single rotation:

1. Single Rotation with left. - Left Rotation(Left of Left)

- The left child (K1) of the root (K2) is promoted as root node.
- The root (K2) is promoted as the right child of the new root(K1).
- The right child of the K1 is converted to the left child of K2



Before Rotation **After Rotation**

Routine to perform single rotation with left

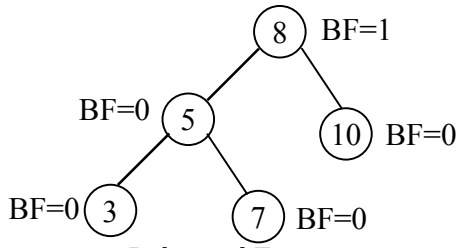
i. This function can be called only if K2 has a left child.
ii. Perform a rotation between node (K2) and its left child (K1)
iii. Update height, then return new root.

```
static Position SingleRotateWithLeft (Position K2)
{
        Position K1;
        K1=K2→left;
        K2→Left = K1→Right;
        K1→Right =K2;
        K2→height= Max (height(K2→left), height (K2→Right))+1;
        K1→height= Max (height(K1→left), (K2→height))+1;
```

```
        return K1;/*New Root*/
    }
```

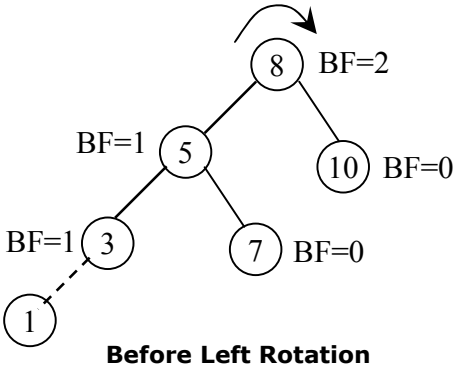
Example:

Consider the following tree which is balanced.

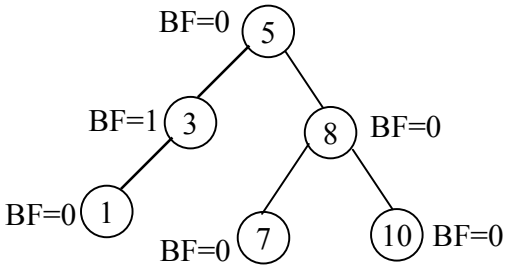


Balanced Tree

Now insert the value '1' it becomes unbalanced due to insertion of the new node in the Left Subtree of the Left Subtree. So we have to make single left rotation in the root node.



Before Left Rotation



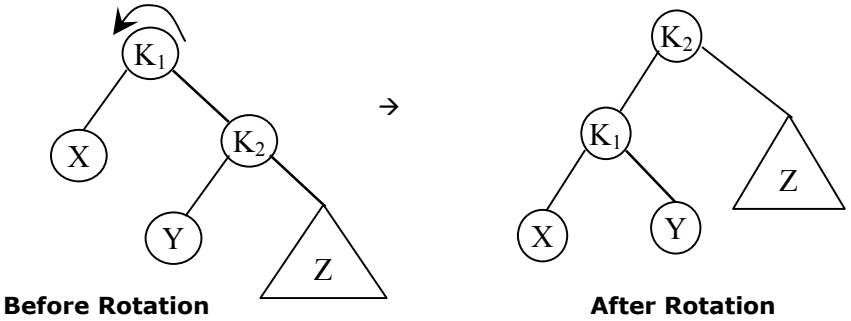
After Left Rotation

2. Single Rotation with Right – Right Rotation(Right of Right)

- The right child(K2) of the root(K1) is promoted as root node.
- Its(K2) left child(Y) is converted to right child of the previous root node(K1).
- The previous root node(K1) is considered as the left child of new root node(K2).

(Algorithm) Routine to Perform Single Rotation with Right:

i. This function can be called only if K1 has a right child.
ii. Perform a rotation between node (K1) and its right child (K2)
iii. Update height, then return new root.



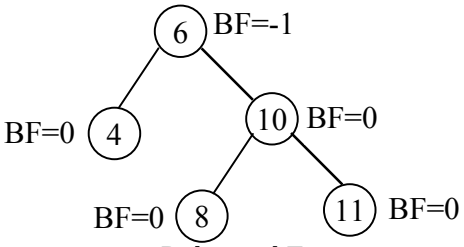
Before Rotation **After Rotation**

```
static Position SingleRotateWithRight (Position K1)
{
        Position K2;
```

```
        K2=K1→Right ;
        K1→Right =  K2→left;
        K2→left =K1;
        K2→height= Max (height(K2→left), height (K2→Right))+1;
        K1→height= Max (height(K→left), (K→height))+1;
        return K2;
    }
```

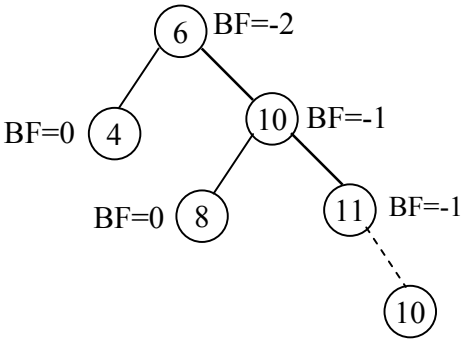
Example:

Consider the following tree which is balanced.

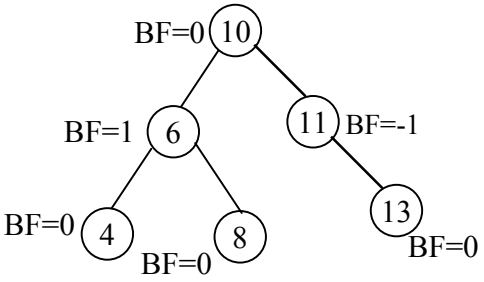


Balanced Tree

Now insert the value '13' it becomes unbalanced due to the insertion of new node in the Right Subtree of the Right Subtree. So we have to make single Right rotation in the root node.



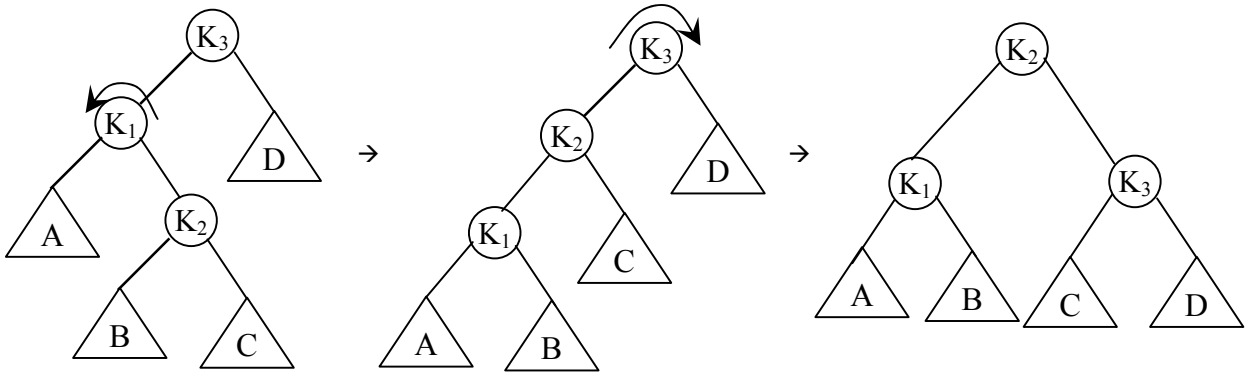
Before Right Rotation



After Right Rotation

Double Rotation:

1. Double Rotation with Left (Insertion in the Right Subtree of the Left Subtree → Left-Right Rotation)



Before Rotation **After Right Rotation** **After Left Rotation**

Routine to perform double rotation with right and then Left

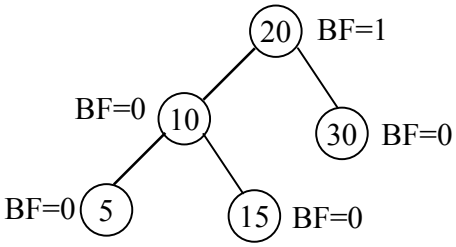
i. Apply single Right rotation at K_1, as a result K_2 becomes the left child of the root node K_3 and K_1 becomes the left child of K_2.

ii. Apply single Left rotation at the root node K_3, as a result K_2 becomes the root node and K_1 becomes the left child of it and K_3 becomes right child of it. Now the tree is balanced.

```
static Position DoubleRotateWithLeft (Position K3)
{
        /* Rotate between K1 and K2*/
        K3→Left = SingleRotateWithRight (K3→Left);
        /* Rotate between K2 and K3*/
        return SingleRotateWithLeft (K3);
}
```

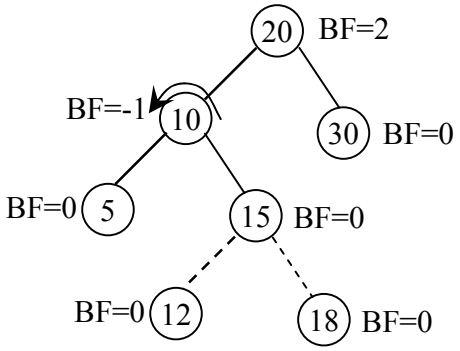
Example:

Consider the following tree which is balanced.

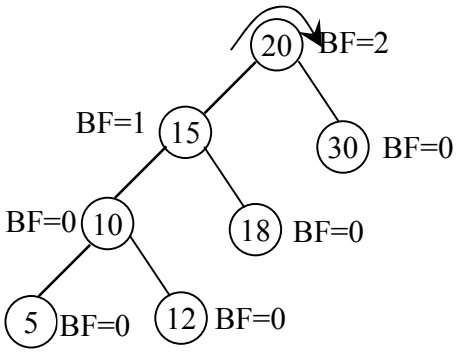


Balanced Tree

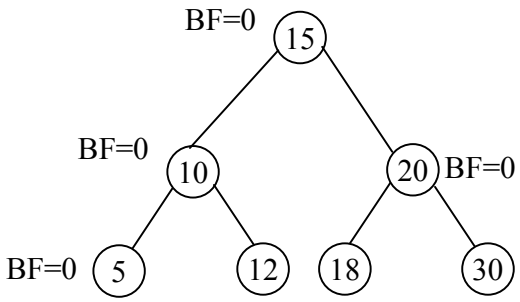
Now insert the value '12'& '18' the tree becomes unbalanced due to the insertion of the new node in the Right Subtree of the Left Subtree. So we have to make double rotation first with right and then with Left.



Before Right Rotation



After Right Rotation & Before Left Rotation



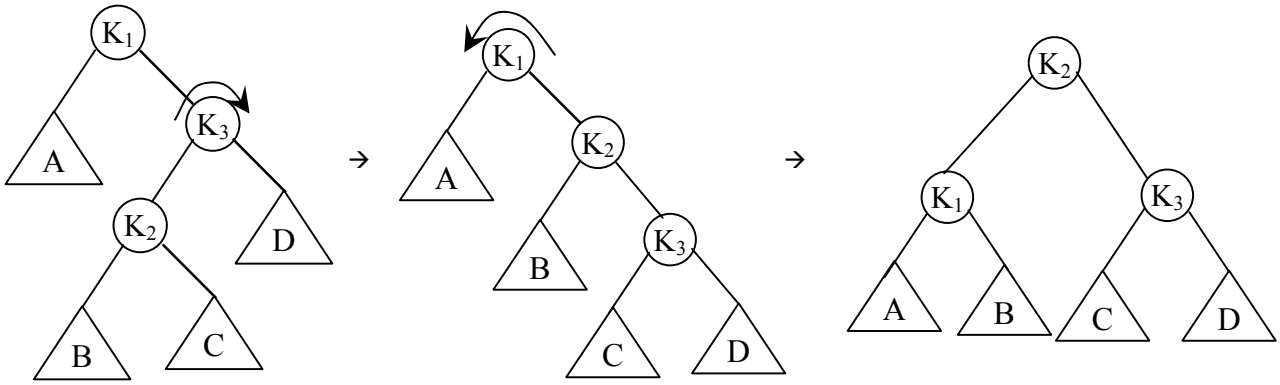
After Left Rotation

2. Double Rotation with Right(Insertion in the Left Subtree of the Right Subtree→Right-Left Rotation)

Routine to perform double rotation with right and then left

i. Apply single Left rotation at K_3, as a result K_2 becomes the right child of the root node K_1 and K_3 becomes the right child of K_2.

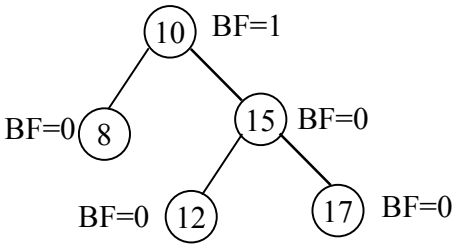
ii. Apply single Right rotation at the root node K_1, as a result K_2 becomes the root node and K_1 becomes the left child of it and K_3 becomes right child of it. Now the tree is balanced.



Before Rotation **After Left Rotation** **After Right Rotation**

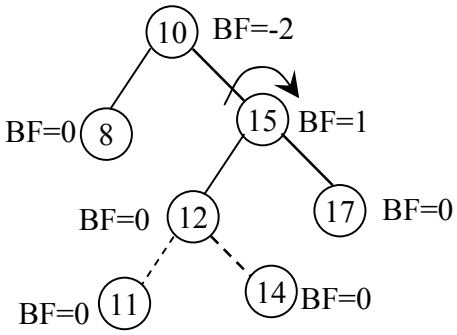
```
static Position DoubleRotateWithRight (Position K1)
{
        /* Rotate between K2 and K3*/
        K→Right = SingleRotateWithLeft (K1→Right);
        /* Rotate between K2 and K1*/
        return SingleRotateWithRight (K1);
}
```

Example:

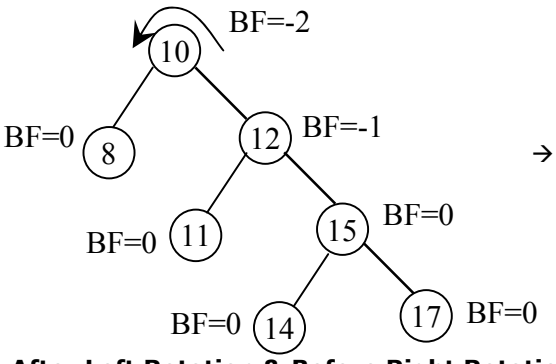


Balanced Tree

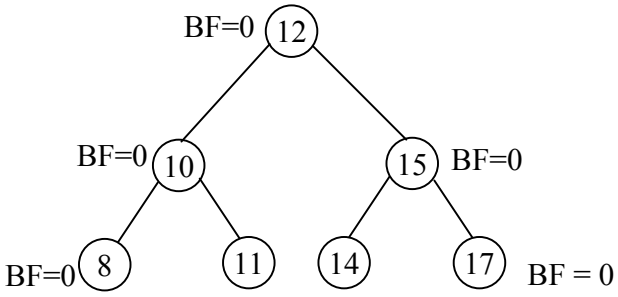
Now insert the value '11'& '14' the tree becomes unbalanced due to the insertion of the new node in the Left Subtree of the Right Subtree. So we have to make double rotation first with Left and then with Right.



Before Left Rotation



After Left Rotation & Before Right Rotation



After Right Rotation

Routine to find height of a node:

```
static int height (Position P)
{
        if (P= =NULL)
                return -1;
        else
                return P→height;
}
```

Declaration for AVL tree

```
Struct AvlNode
{
        DataType data;
        AvlTree Left;
        AvlTree Right;
        int height;
};
```

Algorithm to insert a node in the AVL Tree:

```
AvlTree Insert (DataType X, AvlTree T)
{
        if(T= =NULL)
        {
                T=malloc(size of (struct AvlNode))/*create one node tree*/
                if (T= =NULL)
                        Fatalerror("out of space);
                else
                {
                        T→data =X;
                        T→height=0;
                        T→Left = T→Right = NULL;
                }
        }
        else
        if (X<T→data)
        {
                T→Left = Insert(X,T→Left);
                if (height(T→Left) - height(T→Right)= =2)
                {
                        if (X<T→Left→data)
                                T=SingleRotateWithLeft(T);
                        else
                                T=DoubleRotateWithLeft(T);
```

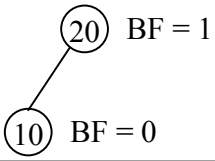
```
                }
        }
        else
        if(X>T→data)
        {
                T→Right=Insert(X,T→Right);
                if (height(T→Right)-height(T→Left)= =2)
                {
                        if (X>T→Right→data)
                                T=SingleRotateWithRight(T);
                        else
                                T=DoubleRotateWithRight(T);
                }
        }
        /else X is in tree already. We will do nothing*/;
        T→height =Max(height(T→Left), height(T→Right))+1;
        return T;
}
```

Example:

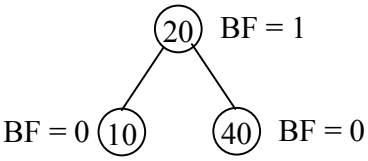
Let us consider how to balance a tree while inserting the numbers 20, 10, 40, 50, 90, 30, 60, 70.

Step 1:(Insert the value 20)

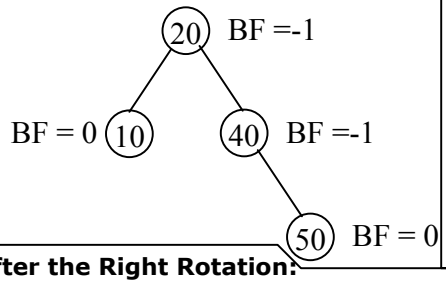
Step 2: (Insert the value 10)



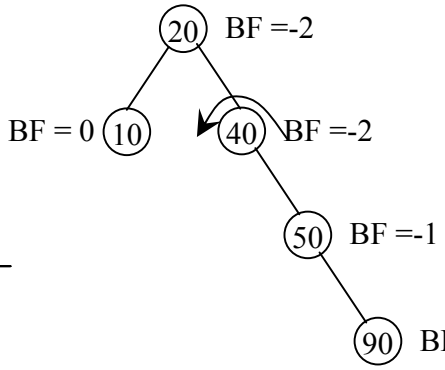
Step 3: (Insert the value 40)



Step 4:(Insert the value 50)

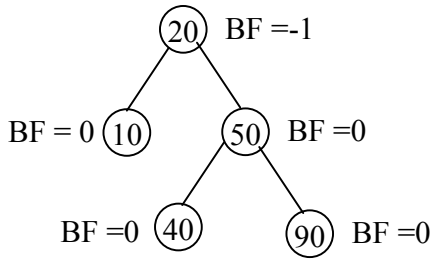


Step 5: (Insert the value 90)



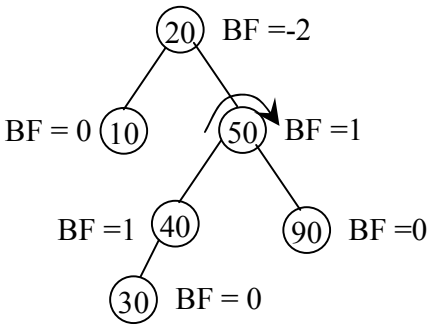
Now the tree is unbalanced due to the insertion of node in the Right subtree of the Right subtree. So we have to make single Right rotation in the node '40'.

After the Right Rotation:



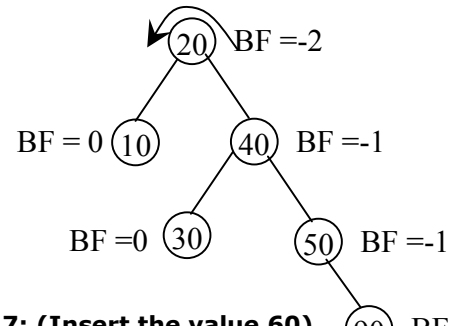
Now the tree is Balanced

Step 6: (Insert the value 30)

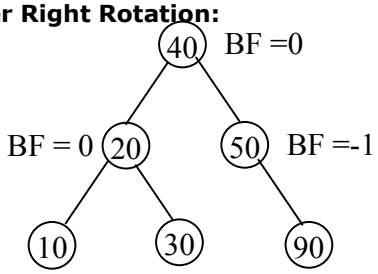


Now the tree is unbalanced due to the insertion of node'30' in the Left subtree of the Right subtree. So we have to make Double rotation first with Left on the node '50' and then with Right on the node '20'.

After Left Rotation:

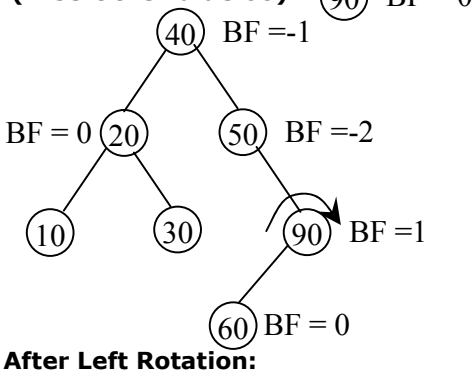


After Right Rotation:



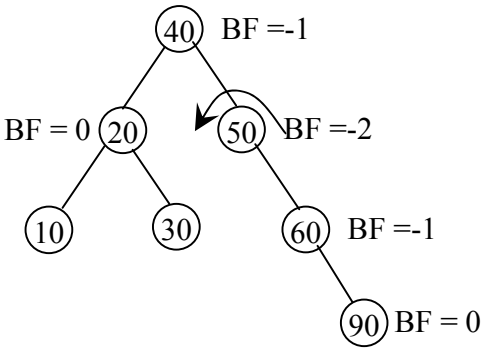
Now the tree is Balanced

Step 7: (Insert the value 60)

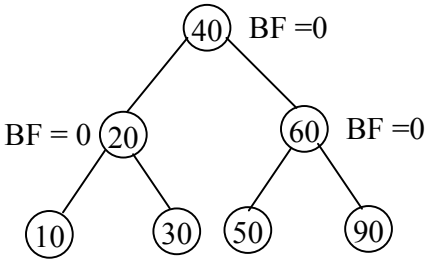


Now the tree at the node '50'is unbalanced due to the insertion of node '60' in the Left subtree of the Right subtree. So we have to make Double rotation first with Left on the node '90' and then with Right on the node '50'.

After Left Rotation:

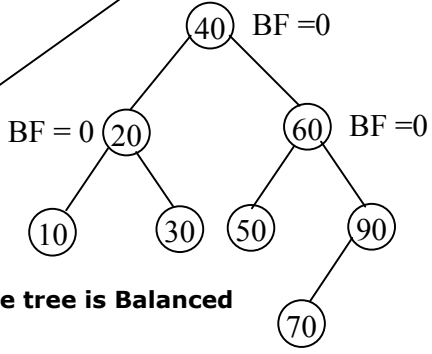


After Right Rotation:



Now the tree is Balanced

Step 8: (Insert the value 70)

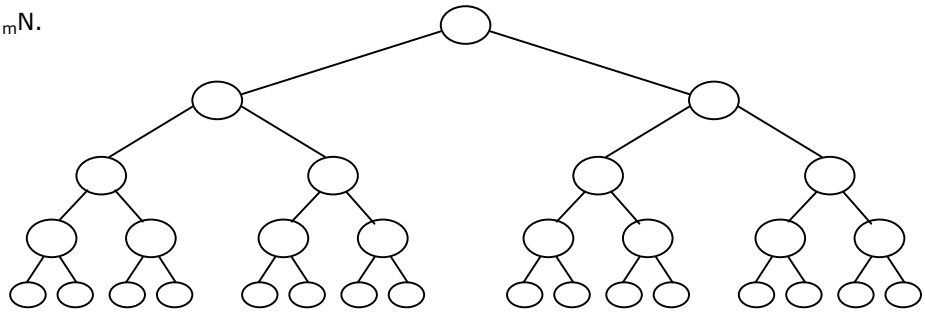


The tree is Balanced

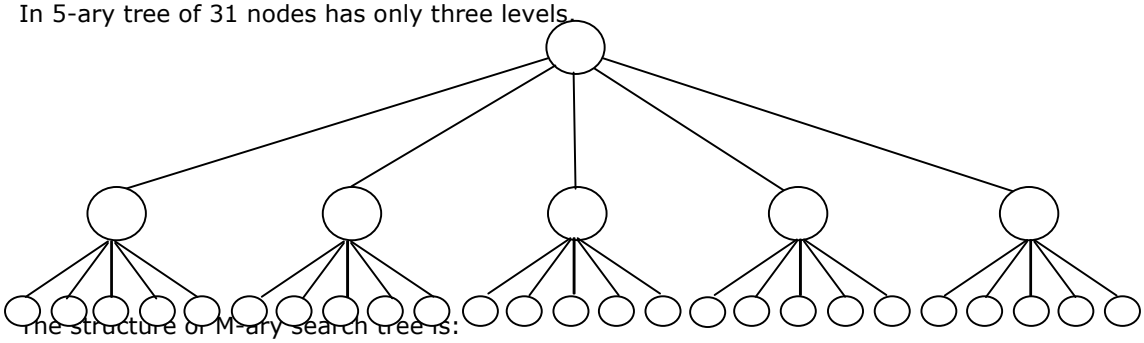
3.2 M- ary search tree B- tree

An M-ary search tree allows. M-way branching. As branching increases, depth decreases. Where as a complete binary tree has height that is roughly equal to $\log_2 N$, a complete M-ary tree has height that is roughly $\log_m N$.

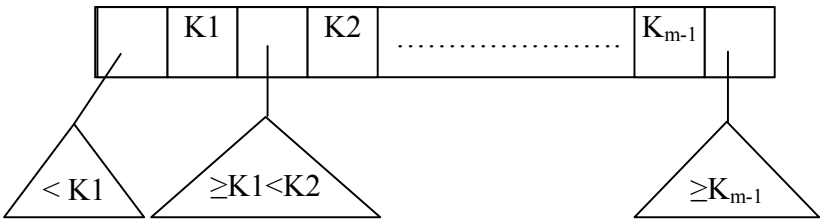
Example:



In 5-ary tree of 31 nodes has only three levels.



The structure of M-ary search tree is:



Each node consists of atmost m-1 keys and m subtrees

Advantages:

- There are limitations on size of the collection that can be maintained in the main memory.
- When data is maintained in the disk, it is important to reduce the number of disc access.
- Many search tree performs this task, by reducing the number of levels.

3.2.1 B tree:

It is a balanced M-ary search tree. A B-tree of order M is an M-ary tree, with the following properties are:

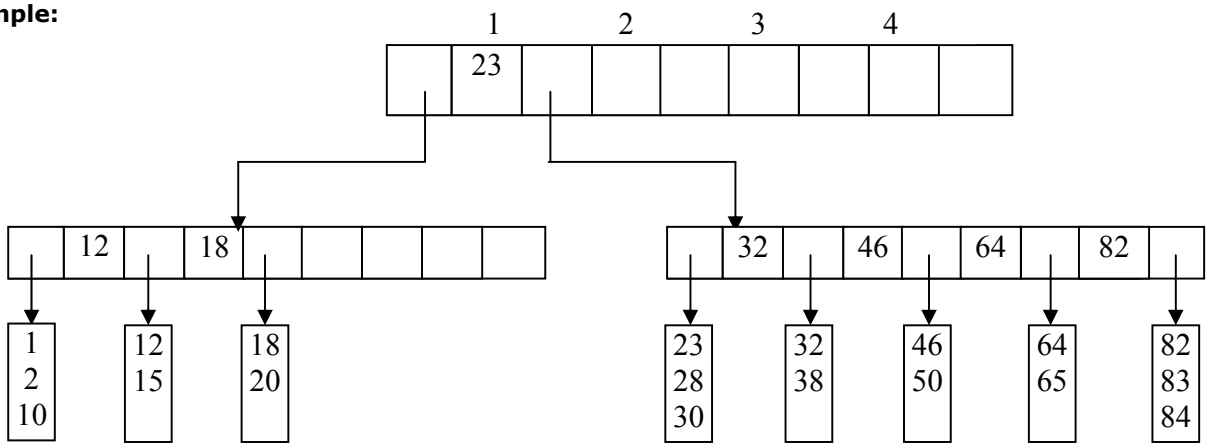
- The data items are stored at levels.
- The non leaf nodes store up to M-1 keys to guide the searching, key i represents the smallest key in subtree i+1.
- The root is either a leaf or has between 2 and M children.
- All non leaf nodes (expect the root) have between M/2 level and M children.
- All leaves are at the same depth and have between L/2 and L children for some L.

Description about B-tree:

- Each node occupies a disk block.
- Interior node stores only the keys which will guide the searching.
- M is determined by size of disk block, size of pointers and number of keys.
- Leaf nodes are different from interior nodes, i.e objects (Data) are stored in leaf nodes only.
- Each leaf can have 'L' objects arranged in sorted order, depends on size of object and disk block.

- All the leaf nodes can be logically organized as one single data file and all other nodes can be organized as a single index files.
- Index file provides a hierarchical index into the database file.
- B-Tree are mostly used for database implementation.
- By using B-Tree efficient search can be done.

Example:

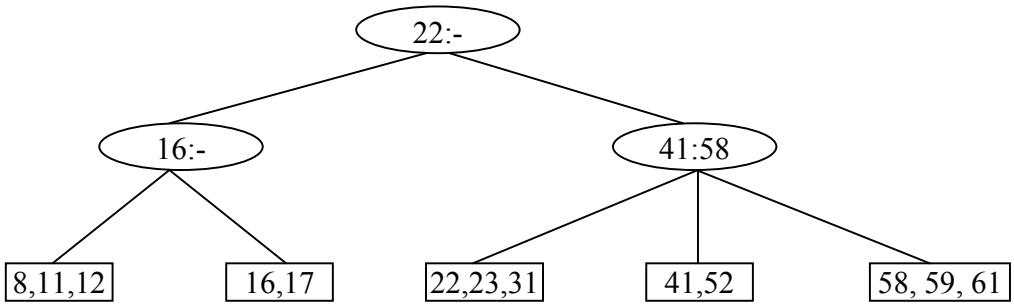


Note:

- Leaf node must have at least ⌊/2 objects
- B-tree of order 4 is known as 2-3-4 tree.
- B-tree of order 3 is known as 2-3 tree,

Representation of 2-3 tree:

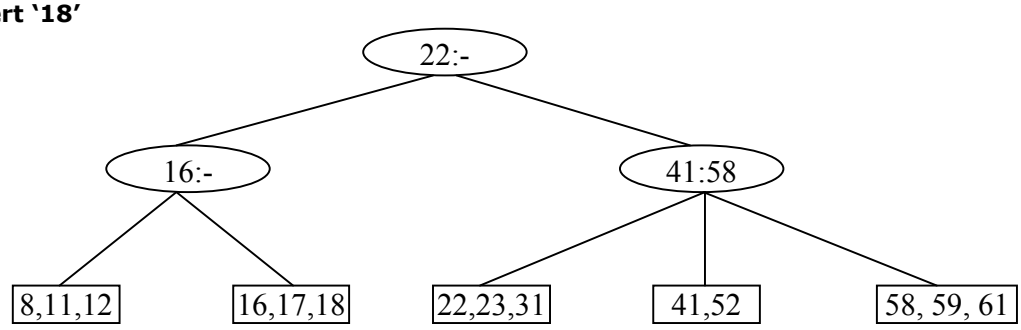
- Interior nodes (non leaves) are represented as ellipses, with 2 pieces of data. Dashed '-----' lines indicates a node with only 2 children.
- Leaves are drawn in boxes, which contains the data in ordered way.



Insertion at 2-3 tree:

To insert the element find the position of insertion in the leaf level.

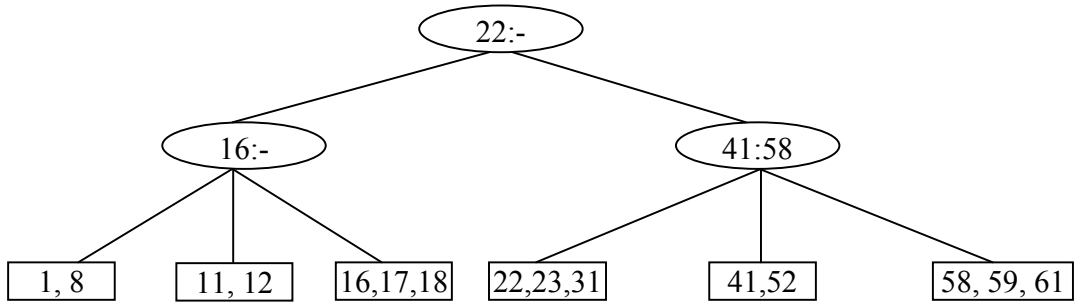
Insert '18'



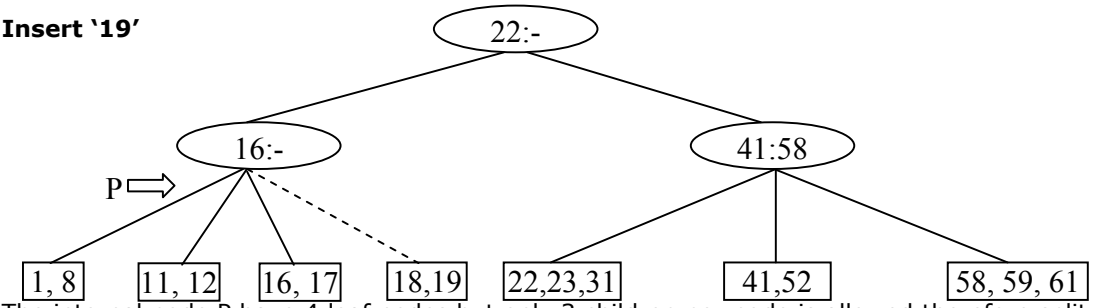
The key '18' is added to leaf level without causing any violations.

Insert '1'

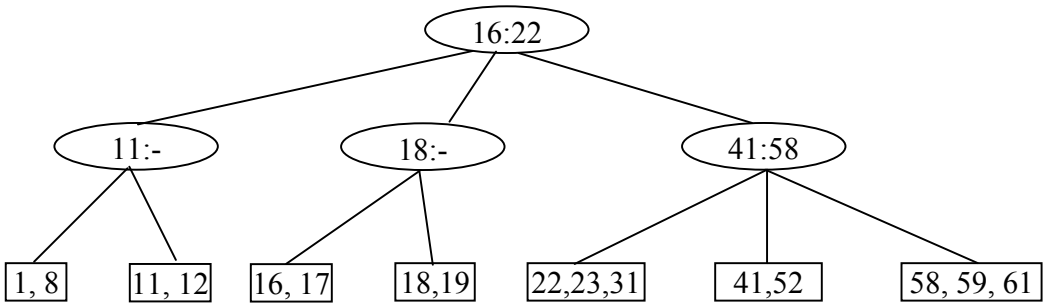
- When '1' is inserted, the leaf node has totally 4 keys. That is 1, 8, 11, 12
- This node violates the condition, since any leaf node can hold only two or three keys. Therefore divide the node into two nodes, with 2 keys in each node



Insert '19'

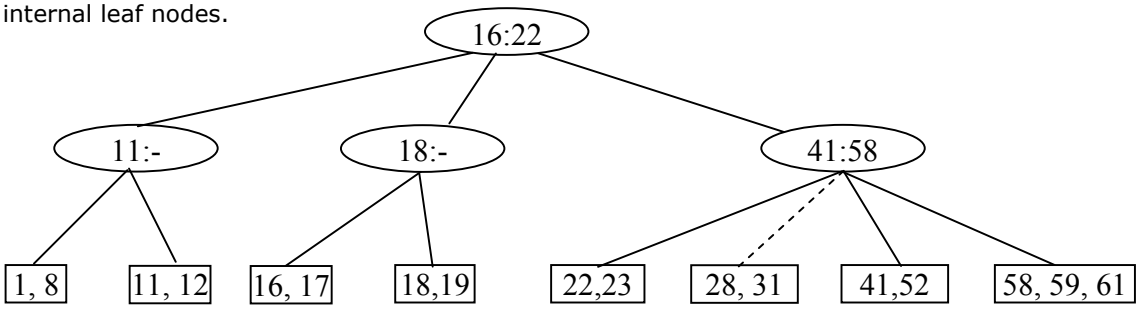


The internal node P have 4 leaf nodes but only 3 children per node is allowed therefore split the node P is into 2 internal node with two children.

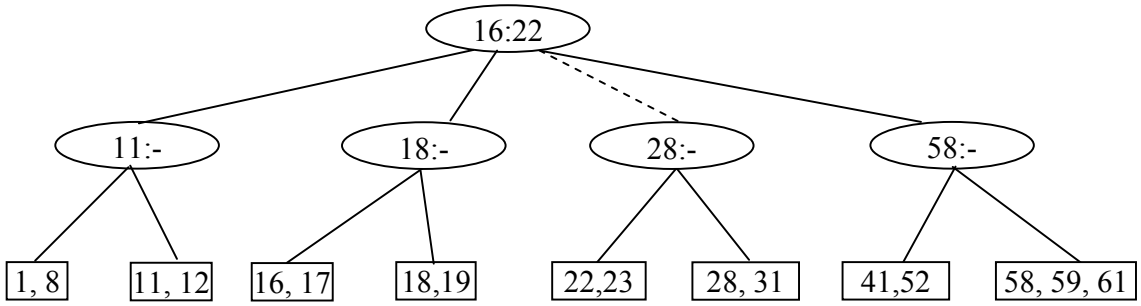


Insert '28':

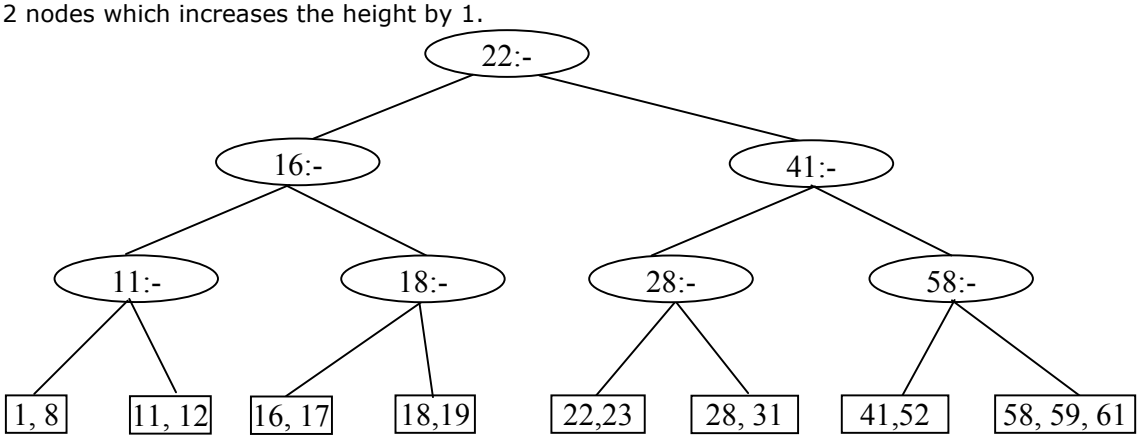
The leaf node 22,23,28,31 has 4 keys, which is not allowed. Therefore split into 2 node and introduce a new internal leaf nodes.



Here the internal node 41:58 has 4 children it violates the rule. So we have to spit it into 2 children.

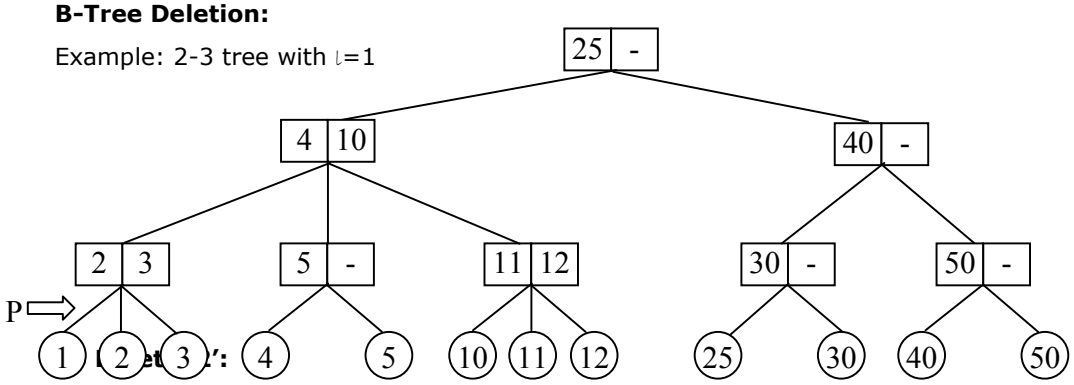


Hence the root has 4 internal nodes in the next level, which violates the rule. Therefore divide the root into 2 nodes which increases the height by 1.



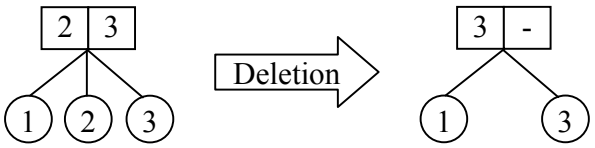
B-Tree Deletion:

Example: 2-3 tree with ⌊=1

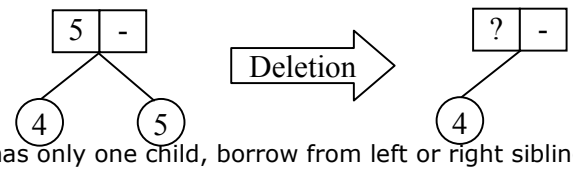


To Delete '2':

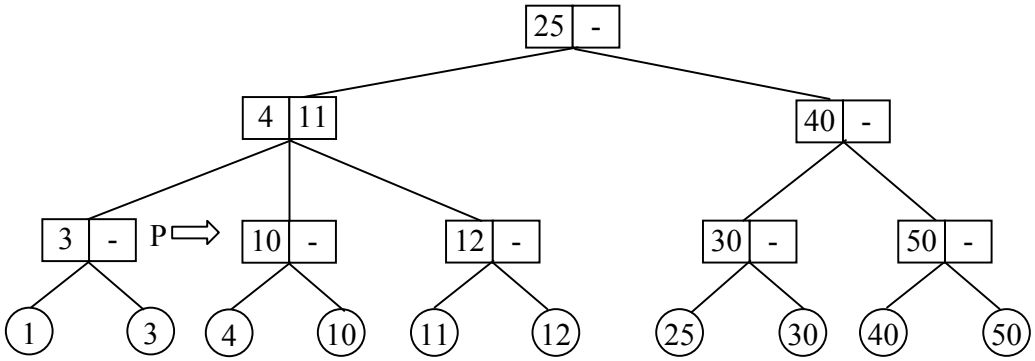
Since P has 3 children's(leaf nodes), '2' can be simply deleted and P be left with just 2 children's.



To Delete '5':

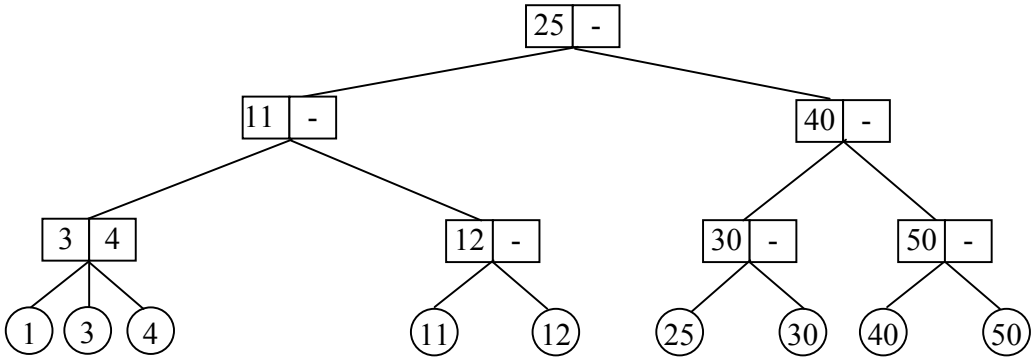


- When the node has only one child, borrow from left or right sibling, without violating the condition.
- Left sibling has least value and the right sibling has greatest value.
- Therefore add '4' to the right sibling.

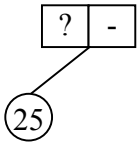


To Delete '10':

- P can't borrow from any of its sibling. So, move the child to left sibling and delete P.

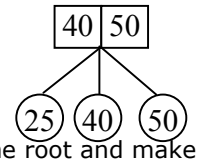


To Delete '30':



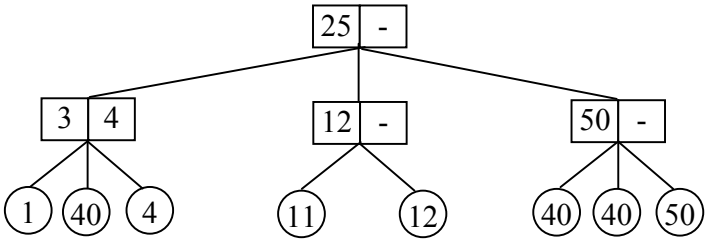
This is invalid

- When the node has only one child, borrow from left or right sibling, without violating the condition.
- Left sibling has least value and the right sibling has greatest value.
- Therefore add '25' to the right sibling.



This is also invalid because the root having only one child. So borrow from left or right sibling without violating the rule.

- So remove the root and make the grant parent as new root.



Algorithm for Searching an element

Step1: Get the searching element x

Step2: Compare x with key of root node

Step3: Find the subtree

Step4: Return the element if it exists in the subtree otherwise return false.

Suppose there are n objects then the tree will have at least $1+\log_3 n$ levels and no more that $1+\log_2 n$. Therefore any access can be perform in O(log n) times.

Application:

- Database implementation
- Indexing on non primary key fields are also possible.

3.3 Priority Queue:

3.3.1 Need for priority queue:

- In a multi user environment, the operating system scheduler must decide which of several processes to run only for a fixed period for time.
- For that we can use the algorithm of QUEUE, where Jobs are initially placed at the end of the queue.
- The scheduler will repeatedly take the first job on the queue, run it until either it finishes or its time limit is up, and placing it at the and of the queue if it doesn't finish.
- This strategy is generally not approximate, because very short jobs will soon to take a long time because of the wait involved to run.
- Generally, it is important that short jobs finish as fast as possible, so these jobs should have precedence over jobs that have already been running.
- Further more, some jobs that are not short are still very important and should also have precedence.
- This particular application seems to require a special kind of queue, known as a PRIORITY QUEUE.

3.3.2 Model:

Priority Queue:

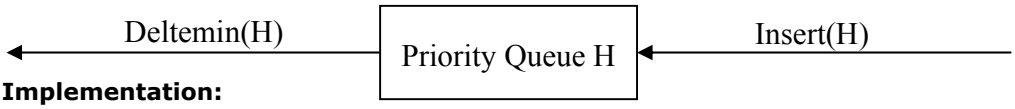
It is a collection of ordered elements that provides fast access to the minimum or maximum element.

Basic Operations performed by priority queue are:

1. Insert operation
2. Deletemin operation

Insert operation is the equivalent of queue's ***Enqueue*** operation.

Deletemin operation is the priority queue equivalent of the queue's ***Dequeue*** operation.



3.3.3 Implementation:

There are three ways for implementing priority queue. They are:

1. Linked list
2. Binary Search tree
3. Binary Heap

Linked list:

A simple linked list implementation of priority queue requires O(1) time to perform the insertion at the front and O(N) time to delete the minimum element.

Binary Search Tree:

- This implementation gives an average running time of O(log N) for both insertion and deletemin operations.

- Deletemin operation makes tree imbalance, by making the right subtree heavy as we are repeatedly removing the node from the left subtree.

3.3.4 Binary Heap:

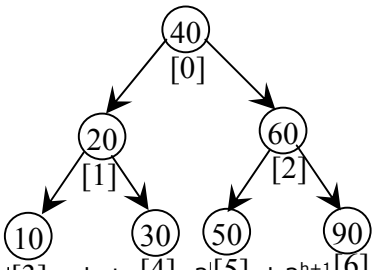
3.3.4.1 Properties:

The efficient way of implementing priority queue is binary heap. Binary heap is merely referred as heaps. Heaps have two properties namely;

1. Structure property
2. Heap order Property

Structure property:

A heap is binary tree that is fully completely filled, with the possible exception of the bottom level, which is filled from left to right. Such a tree is known as a *complete binary tree*.



A complete binary tree of height has between 2^{h} and $2^{h+1}-1$ nodes (i.e if the tree has height h=1, then number of nodes it have must be 2 or 3). This implies that the height of complete binary tree is log N which is clearly O(log N).

An array implementation of complete binary tree is as follows:

40	20	60	10	30	50	90
[0]	[1]	[2]	[3]	[4]	[5]	[6]

For any element in array position i:

i) the left child is 2i + 1

ii) the right child is in the cell after the left child 2i+2

iii) the parent is in position [(i+1)/2]

Example:

If i = 0,

i) the parent node is [(i+1)/2] = ½ = 0.5 = app 1 i.e it refers the root node '40'

ii) the left child is 2i +1 = (2*0)+1 = 1 i.e it refers the position of the element '20'

iii) the right child is 2i+2 = (2*0)+2 = 2 i.e it refers the position of the element '60'

If i = 1,

i) the parent node is [(i+1)/2] = 2/2 = 1 i.e it refers the root node '20'

ii) the left child is 2i+1 = (2*1)+1 = 3 i.e it refers the position of the element '10'

iii) the right child is 2i+2 = (2*1)+2 = 4 i.e it refers the position of the element '30'

If i = 2,

i) the parent node is [(i+1)/2] = 3/2 = 1.5 = app 2 i.e it refers the root node '60'

ii) the left child is 2i+1 = (2*2)+1 = 5 i.e it refers the position of the element '50'

iii) the right child is 2i+2 = (2*2)+2 = 6 i.e it refers the position of the element '90'

Binary heaps is represented as array it doesn't require pointers and also the operations required to traverse the tree are extremely simple and fast. But the only disadvantage is to specify the maximum heap size in advance.

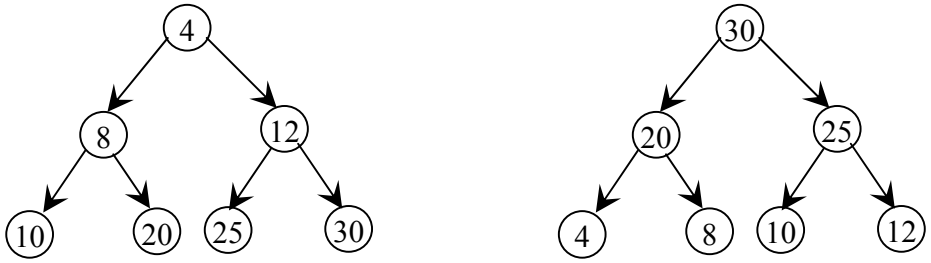
Heap Order Property:

In a Heap, for every node X, the key in the parent of X is smaller than (or equal to) the key in X, with the exception of the root(which has no parent).

The Heap order property varies for the two different types of heaps,

1) MinHeap
2) MaxHeap

In a MinHeap, for every node X, the key in the parent of X is smaller than (or equal to) the key in X, with the exception of the root(which has no parent). [figure 1]



In a MaxHeap, for every node X, the key in the parent of X is greater than (or equal to) the key in X, with the exception of the root(which has no parent). [figure 2]

Routine for the declaration of priority Queue

```
Struct heap
{
        int capacity;
        int size;
        ElementType  *Element;
};
```

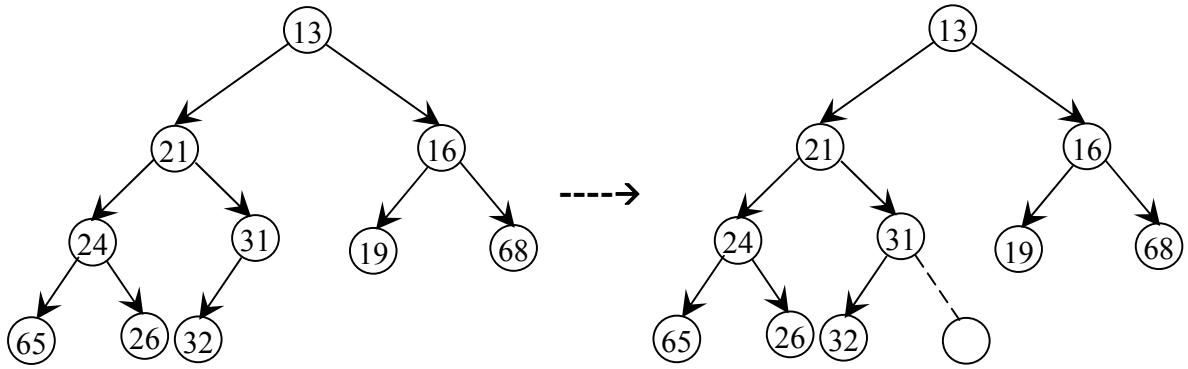
3.3.4.2 Basic heap operations

Insertion

- To insert an element **X** into the heap, create a hole in the next available location, since otherwise the tree will not be complete.
- If **X** can be placed in the hole without validating the heap order property, then we can insert.
- Otherwise we slide the element is in the *holes parent node* in to the hole, thus bubbling the hole up towards the root. We continue this property until the element **X** can be placed in the hole.

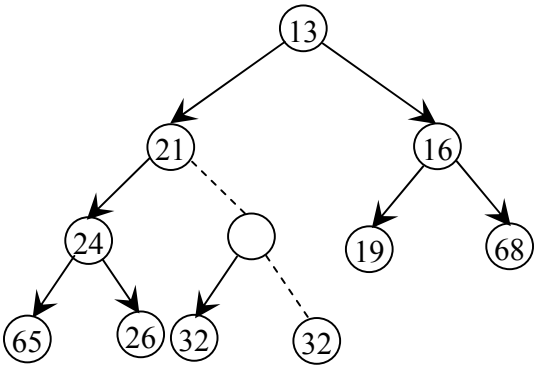
To insert '14'

Step1: Create a hole in the next available location, since otherwise the tree will not be complete.



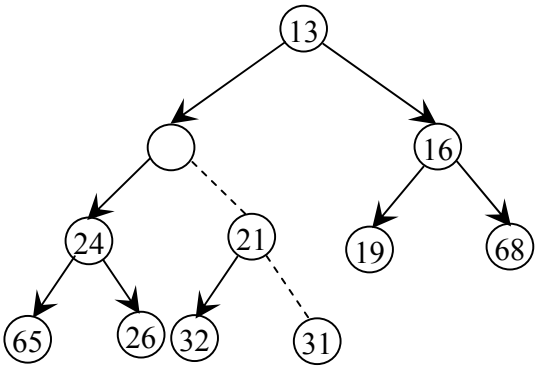
Step2: Check the inserting element '14' with the holes parent node '31'.

- If the inserting element is greater than the holes parent node element, then the element will be inserted in to the hole.
- If the inserting element is lesser than the holes parent node element (14<31), we have to slide the holes parent node element in to the hole. Thus, the bubbling the hole up towards the root happen.

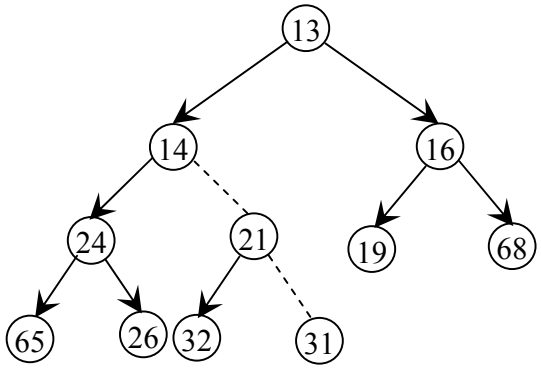


Step3: Repeat the step 2 check the inserting element '14' with the holes parent node '21'.

- If the inserting element is greater than the holes parent node element, then the element will be inserted in to the hole.
- If the inserting element is lesser than the holes parent node element (14<21). We have to slide the holes parent node element in to the hole. Thus the bubbling the hole up towards the root happens.



Step4: Inserting the element '14' in to the hole will not validate heap order Property. Therefore you can insert the element '14' in to he hole.



This general strategy is known as a perculate up; the new element is perculated up the Heap until the correct location is found.

Perculation in the *Insert* routine by performing repeated swaps until the correct order was established.

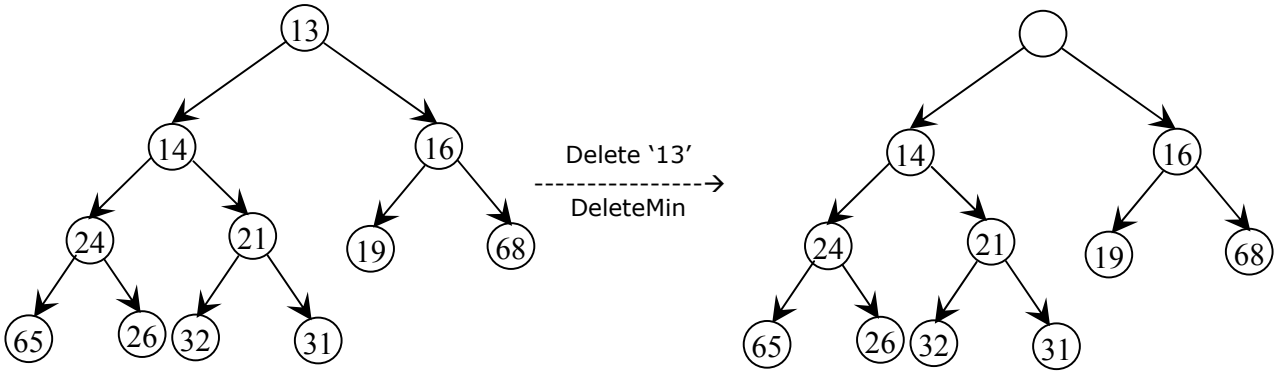
The time to do the insertion could be O(log N).

Routine to insert in to a binary heap

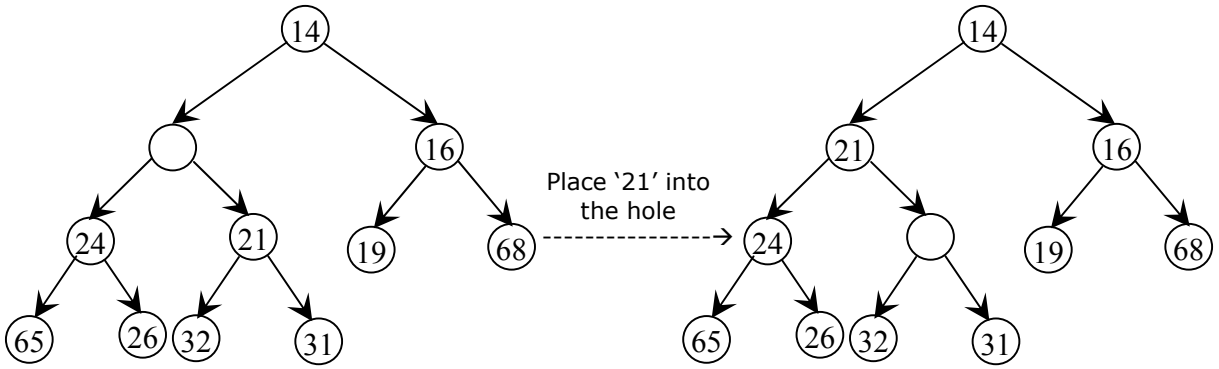
```
void Insert (Element Type X, PriorityQueue H)
{

        int i;
        if (IsFull(H))
        {
                Error ("priority Queue is full");
                return;
        }
        for (i=++H→size; H→Element [i/2]>X; i/2)
        {
                H→Element[i] = H→Element[i/2];
        }
                H→Element[i]= X;
}
```

DeleteMin:

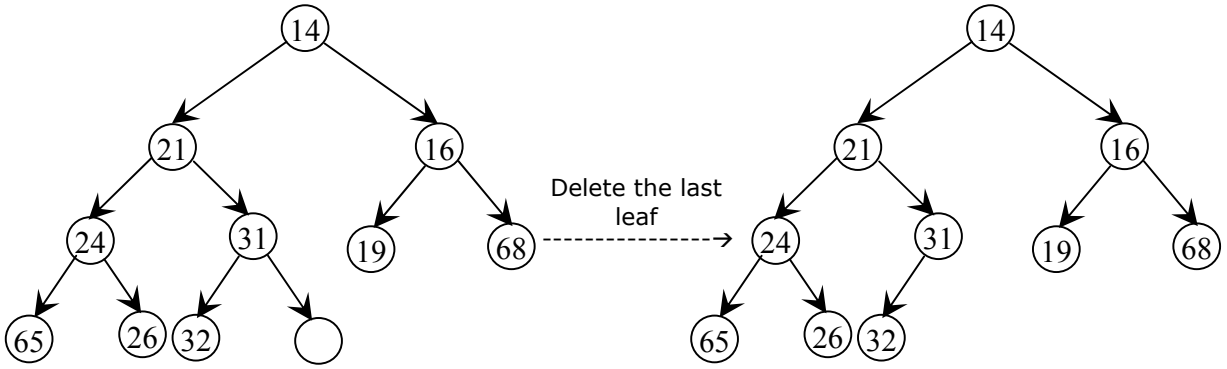
- Finding the minimum is easy, it is hard to move. When the minimum is removed a hole is created at the root.
- Since the Heap now becomes one smaller, it follows that the last element X in the Heap must move somewhere in the Heap.
- If X can be placed in the hole, then we are done.
- We slide the smaller of the hole's children into the hole, thus pushing the hole down one level.
- We repeat this step until X can be placed in the hole.
- Finally, Delete the last leaf.



- The last value cannot be placed in the hole, because this would violate heap order.
- Therefore, we place the smaller child '14' in the hole, sliding the hole down one level.



Now slide the hole down by inserting the value '31' in to the hole.



Routine to DeleteMin in the Binary Heap

```
Element type deletemin (Priority Queue H)
{
        int i, child;
        ElementType MinElement, LastElement;
        if(IsEmpty(H)
        {
                Error ("PRIORITY QUEUE is empty");
                return H→Element[0];
        }
        MinElement = H→Element[1];
```

```
                LastElement = H→Element[H→Size--];
                for (i=1;i*2<=H→size; i=child)
                {
                        /*Finding Smaller Child*/
                        child=i*2;
                        if(child!=H→size &&H→Element[child]>H→Element[child+1])
                                child++;
                        /*percolate one level*/
                        if (LastElement>H→Element[child];
                        else
                                break;
                }
                H→Element[i] = LastElement;
                return MinElement;
        }
```

Application Priority Queues:

1. Selection Problem
2. Event Simulation

3.4 Hashing

- Hashing is a key to address transformation technique.
- The hash table or hash map is a data structure that associates key with value.
- The implementation of hash tables is frequently called hashing.
- Hashing is used for faster access of elements and records from the collection of tables and files.
- Hashing is a technique used for performing insertions, deletions and search operations in constant average time of (1).
- Hashing is applied where the array size is large and time taken for searching an element is more.
- It works by transforming the key using a hash function into a hash, a member which is used to index into an array to locate the desired location where the values should be.
- Hash table supports the efficient addition of new entries and the time spent searching for the required data is independent of the number of items stored.

3.4.1 Hash table:

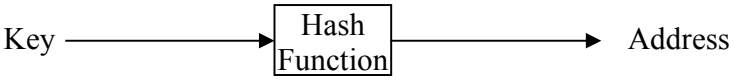
- In hashing an ideal hash table data structure is nearly an array of fixed size, containing the key.
- The key is a string or integer associate with a value. Each key is mapped in to some number in the range 0 to TableSize -1 and placed in the appropriate cell.
- The mapping is called hash function, which should be simple to compute and should ensure that any two distinct keys get different cells.

```
0
1
2
3       A
```

4	B
5	
6	C
7	
8	
9	

3.4.2 Hash function:

- Hash function is a key to address transformation which acts upon a given key to compute the relative position of the key in an array.
- The mapping of each key into some number ranged from 0 to TableSize-1 is known as Hashing.
- Ideally, the hash function is used to determine the location of any record given its key value.
- The hash function should have the following properties
  - Simple to compute
  - Must distribute the data evenly.
  - Generates lower number of collisions.
  - Reduce storage requirement.
- Hashing is a method to transform a key to an address. The transformation involves application of a function to the key value, as shown below.



- The hash function in other words, maps a key value to an address in the table and the domain of the hash function is non-negative integers in the range 0 to the size of the table.
- If x is the key and h is the hash function, then h(x) is the address for the key to be stored.

Simple Hash function:

Consider a simple hash function, h(x) = x mod 10. The following table shows list of key values and the corresponding hash function values.

Key (x)	Hash Function Value h(x)
10	0
25	5
33	3
47	7
64	4
88	8
39	9

The following figure shows the Storage of the key values in the above table using the hash function value as index.

Key (x)	Hash Function Value h(x)
0	10
1	
2	

3	33
4	64
5	25
6	
7	47
8	88
9	39

Suppose we want to store valu 25, we need to apply the hash function first.

In this case, it is 25 mod 10 = 5 . 5 is used as the address for storing 25

- The simple hash function is *HashValue = key % TableSize*.

Routine for simple Hash Function

```
Typedef unsigned int index;
index Hash (const int*key, int TableSize)
{
        index HashVal =0;
        While(*key !='\0')
        HashVal+=*key++;
        return HashVal % TableSize ;
}
```

- For instant suppose the table size is 10007. The given input key can have maximum of 8 character are few characters long.
- Since a char as an integer value (i.e) always atmost 127 (i.e) ASCII code can represent in 7 bits therefore 2^7=128.
- The hash function can 1st assume the values between 0 to 1016 (i.e) 127x8. Hence it cannot provide equal distribution.

Routine for Second Hash function:

```
Typedef unsigned int index;
index Hash (const char*key, int TableSize)
{
        return (key[0]+27*key[1]+729*key[2])% TableSize;
}
```

- In this hash function we can assume at least 2 characters + null terminator. The value 27 represents number of English alphabets 26+a blank space. Therefore 729 is 27^2.
- This function examines only the first 3 char, if these are random and the table size is 10007, (i.e) English is not random. Therefore 26^3=17576 possible combination of 3 char ignoring blank space.

Routine for Third Hash function:

```
Typedef unsigned int index;
index Hash (const char*key,int TableSize)
{
        index HashVal=0;
```

```
        While(*key='\0')
        HashVal=(Hash<<5)+*key++;
        return HashVal % TableSize;
    }
```

3.4.3 Collision:

- When a memory location filled if another value of the same memory location comes then there occurs collision.
- When an element is inserted it hashes to the same value as an already inserted element, then it produces collision and need to be resolved.

Collision resolving methods:

Separate chaining (or) External Hashing

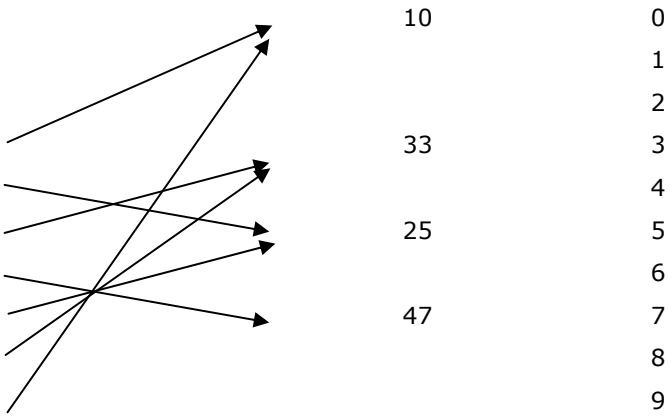
Open addressing (or) Closed Hashing

1. Separate chaining (or) external hashing:

- Separate chaining is a collision resolution technique, in which we can keep the list of all elements that hash to same value. This is called as separate chaining because each hash table element is a separate chain (linked list).
- Each link list contains the entire element whose keys hash to the same index.

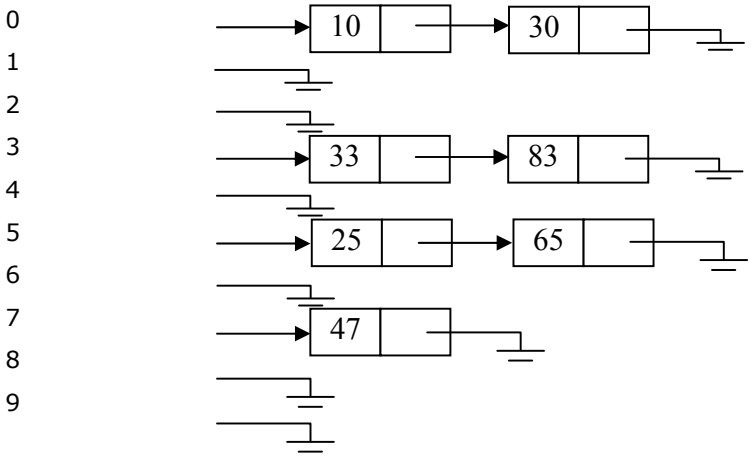
Collision diagram:

Key (x)	Hash Function Value h(x)
10	10 % 10 = 0
25	25 % 10 = 5
33	33 % 10 = 3
47	47 % 10 = 7
65	65 % 10 = 5
83	83 % 10 = 3
30	30 % 10 = 0



Hash Table

- Separate chaining is an open hashing technique.
- Each linked list contain all the elements whose keys to the same index.
- A pointer field is added to each record location.
- When an overflow occurs this pointer is set to point the overflow blocks making a linked list.
- In this method, the table can never overflow, since the linked lists are only extended upon the arrival of new keys.
- The following figure shows separate chaining hash table.



List_Node → Structure is the same as the linked list declaration

Hash_Table → Structure contains an array of linked lists, which are dynamically allocated when the table is initialized.

The_List → Pointer to a pointer to a List_Node structure.

Type declaration for open hash table

```
typedef struct List_Node *Node_Ptr;
struct List_Node
{
        Element_Type element;
        Node_Ptr next;
};
typedef Node_Ptr List;
typedef Node_Ptr Position;

/* LIST *the_list will be an array of lists, allocated later */
/* The lists will use headers, allocated later */
struct Hash_Tbl
{
        unsigned int Table_Size;
        List *The_Lists;
};
typedef struct Hash_Tbl *Hash_Table;
```

Initialization routine for open hash table

```
Hash_Table Initialize_Table( unsigned int Table_Size )
{
        Hash_Table H;
        int i;
        if(Table_Size < MIN_TABLE_SIZE )
        {
                error("Table size too small");
```

```
                return NULL;
        }
        /* Allocate table */
        H = (Hash_Table) malloc ( sizeof (struct Hash_Tbl) );
        if( H == NULL )
                fatal_error("Out of space!!!");
        H->Table_Size = next_prime(Table_Size );
        /* Allocate list pointers */
        H->The_Lists = (position *) malloc( sizeof (LIST) * H-> Table_Size);
        if( H->The_Lists == NULL )
                fatal_error("Out of space!!!");
        /* Allocate list headers */
        for(i=0; i<H->Table_Size; i++ )
        {
                H->The_Lists[i] = (LIST) malloc( sizeof (struct List_Node) );
                if( H->The_Lists[i] == NULL )
                        fatal_error("Out of space!!!");
                else
                        H->The_Lists[i]->next = NULL;
        }
        return H;
    }
```

Find Function:

1. Use the hash function to determine which list to traverse
2. Traverse the list in normal manner.
3. Return the position where the item is found.
4. The call Find(Key, H) will return a pointer to the cell containing key.
5. If Element_Type is a string, comparison and assignment must be done with strcmp & strcpy respectively.

Find routine for open hash table

```
Position Find( Element_Type Key, Hash_Table H )
{
        Position p;
        List L;
        L = H->The_Lists[ Hash( Key, H->Table_Size) ];
        P = L->next;
        while( (P != NULL) && (P->element != Key) )
        /* Probably need strcmp!! */
        P = P->next;
        return P;
}
```

Insert Function:

1. Go to the position by the hash function for the item X.
2. Traverse the list to see if X exists already.
3. It not, insert a new node at the rear of the list.

Insert routine for open hash table

```
Void Insert( Element_Type Key, Hash_Table H )
{
        Position Pos, New_Cell;
        List L;
        Pos = Find( Key, H );
        if( Pos == NULL )
        {
                New_Cell = (Position) malloc(sizeof(struct List_Node));
                if( New_Cell == NULL )
                        fatal_error("Out of space!!!");
                else
                {
                        L = H->The_Lists[ Hash( Key, H->Table Size ) ];
                        New_Cell->next = L->next;
                        New_Cell->element = Key; /* Probably need strcpy!! */
                        L->next = New_Cell;
                }
        }
  }
```

Advantages:

i) More number of elements can be inserted as it uses array of linked list.

ii) The elements having the same memory address will be in the same chain and hence leads to faster searching.

iii) Doesn't require a prior knowledge of the number of elements that are to be stored in the hash table (i.e.) dynamic allocation is done.

iv) It helps to get a uniform and perfect collision resolution hashing.

Disadvantages:

i) The elements are not evenly distributed. Some hash index may have more elements and some may not have anything.

ii) It requires pointers which require more memory space. This leads to slow the algorithm down a bit because of the time required to allocate the new cells, and also essentially requires the implementation of a second data structure

2. Open addressing or Closed Hashing:

❖ Open addressing hashing is an alternating technique for resolving collisions with linked list. In this system if a collision occurs, alternative cells are tried until an empty cell is found.

- The cell $h_0(x)$, $h_1(x)$, $h_2(x)$……. Are tried in succession, where $h_i(x)$ = (Hash (X) + F(i)) mod Table_Size with F(0) = 0.
- The function F is the collision resolution strategy. This technique is generally used where storage space is large.
- Arrays are used here as hash tables.

Definition: The technique of finding the availability of another suitable empty location in the hash table when the calculated hash address is already occupied is known as open Addressing. There are three common collisions resolving strategic.

1. Linear probing
2. Quadratic Probing
3. Double hashing

Linear probing:

- Probing is the process of a placing in next available empty position in the hash table. The Linear probing method searches for next suitable position in a linear manner(next by next). Since this method searches for the empty position in a linear way, this method is referred as linear probing.
- In linear probing for the i^{th} probe, the position to be tried is, (h(k) + i) mod Table_Size, where 'f' is a linear function of i, F(i)=i. This amounts to trying cells sequentially in search of an empty cell.

Example for Linear Probing:

Insert the keys 89, 18, 49, 58, 69 into a hash table using in the same hash function as before and the collision resolving strategies. F(i)=i.

Solution:

- In this e.g initially 89 is inserted at index '9'. Then 18 is inserted at index 8.
- The first collision occurs when 49 is inserted. It is put in the next available index namely '0' which is open.
- The key 58 collides with 18, 89, 49 afterwards it found an empty cell at the index 1.
- Similarly collision 69 is handled.
- If the table is big enough, a free cell can be always be found, but the time to do so can get quite large.
- Even if the table is relatively empty, blocks of occupied cells start forming. This is known as primary clustering means that any key hashes into the cluster will require several attempts to resolve the collision and then it will add to the cluster.

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						

7					
8		18	18	18	18
9	89	89	89	89	89

Algorithm for linear probing:

1. Apply hash function on the key value and get the address of the location.
2. If the location is free, then
   i) Store the key value at this location, else
   ii) Check the remaining locations of the table one after the other till an empty location is reached. Wrap around on the table can be used. When we reach the end of the table, start looking again from the beginning.
   iii) Store the key in this empty location.
3. End

Advantages of linear probing:

1. It does not require pointers.
2. It is very simpler to implement.

Disadvantages of linear probing:

1. It forms clusters, which degrades the performance of the hash table for sorting and retrieving data.
2. If any collision occur when the hash table becomes half full, it is difficult to find an empty location in the hash table and hence the insertion process takes a longer time.

UNIT IV: GRAPHS

Definitions – Topological sort – breadth-first traversal - shortest-path algorithms – minimum spanning tree – Prim's and Kruskal's algorithms – Depth-first traversal – biconnectivity – euler circuits – applications of graphs

Unit IV

Representation of Graph

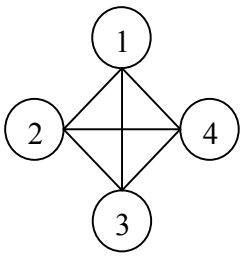
- Graph terminology including these terms: Vertex, edge, adjacent, incident, degree, cycle, path, connected component, and spanning tree.
- Three types of graphs : Undirected, directed, and weighted
- Common graph representations: adjacency matrix, packed – adjacency lists, and linked – adjacency lists.
- Standard graph search methods: breadth-first and depth- first search.
- Algorithms to find a path in a graph to find the connected components of an undirected graph, and to find a spanning tree of a connected undirected graph.
- Specifying an abstract data type as an abstract class.

4.1 DEFINITIONS: A *graph* G= (V, E) is an ordered pair of finite sets V and E. The elements of V are called as vertices are also called as nodes and points. The elements of E are called edges are also called arcs and lines. Each edge in E joins two different vertices of V and is denoted by the tuple (i,j), where i and j are the two vertices joined by E.

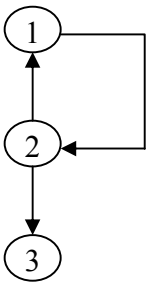
4.1.1 Graph Display: A graph generally represented as figure in which the vertices represented by circles and the edges by lines. Examples, of graphs are as follows:

Some of the edges in this figure are oriented (that they have arrow heads) while other are not. An edge with an orientation is a directed while an edge with no orientations is an undirected edge. The undirected edges (i,j) and (j,i) are the same; the directed edge (i,j) is different from the directed edge(j,i) the former being oriented from I to j and the latter from j to i.

Example: -



Undirected Graph G1



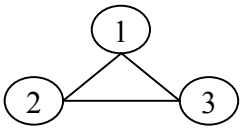
Directed graph G2

V (G1) = { 1, 2, 3, 4}; E (G1) = { (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4) }

V (G2) = { 1, 2, 3 }; E (G2) = { < 1, 2 >, < 2, 1 >, < 2, 3 > }

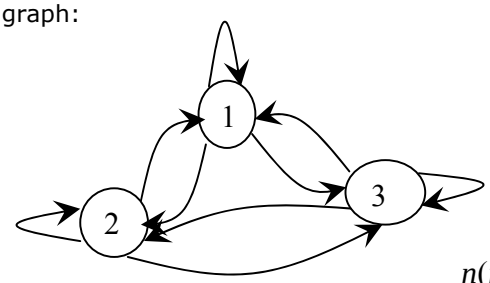
The edges of a directed graph are drawn with an arrow from the tail to the head.

For undirected graph:



$$\frac{n(n-1)}{2} = 3 \; edges$$

For directed graph:

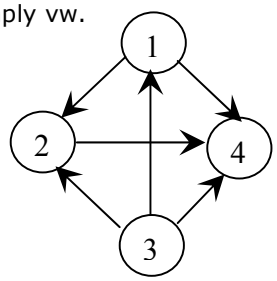


n^2 = 9 edges

The no.of possible edges in an undirected graph is $\frac{n(n-1)}{2}$ and in a directed graph is n^2.

4.1.2 Directed Graph:

A directed graph or di-graph is a pair G = (V,E) where V is a set whose elements are called vertices(nodes) and E is a set of ordered pairs of elements of V called edges or directed edges or arcs. For directed edges(v,w) in E, 'v' is its tail and 'w' is its head. (v,w) is represented in the diagrams as the arrow, v→w, i.e simply vw.

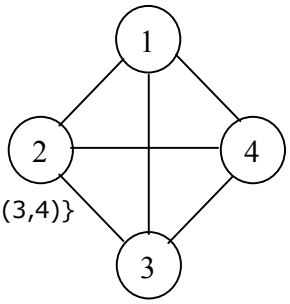


Vertices = {1,2,3,4}

Edges = {(1,2), (1,4), (2,4), (3,1), (3,2), (3,4)}

4.1.3 Undirected Graph:

An undirected graph is a pair G = (V,E), where V is a set whose elements are called vertices(nodes) and E is a set of unordered pairs of distinct elements of V called edges or undirected edges. Each edge may be considered as a subset of V containing two elements, {v,w} denotes an undirected edge, represented as v-w i.e simply vw . Of course vw = wv for undirected graph.



Vertices = {1,2,3,4}

Edges = {(1,2), (1,4), (1,3), (2,4), (2,3), (3,4)}

Note:

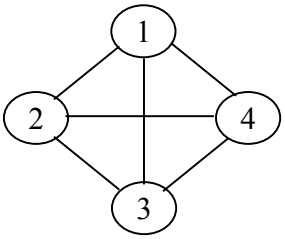
i) An E set cannot have duplicate elements.

ii) An edge that connects a vertex to itself is not possible.

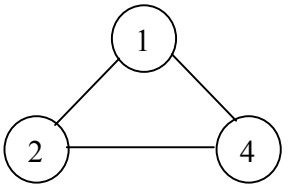
4.1.4 Subgraph:

A subgraph of graph G = (V,E) is a graph G'=(V',E') such that V'⊆V and E'⊆E.

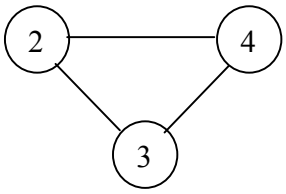
Graph G1



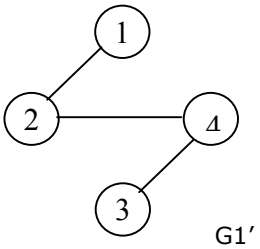
Some of the subgraphs for the graph G1:



G1'



G1'

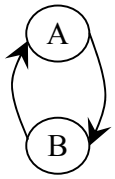


G1'

4.1.5 Symmetric Digraph:

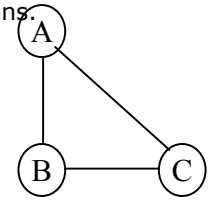
A symmetric digraph is a directed graph such that for every edge vw there is also the reverse edge wv.

Example:



4.1.6 Symmetric Undirected graph:

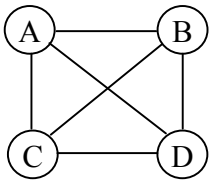
Every undirected graph is a symmetric digraph by interpreting each undirected edge as a pair of directed edges in opposite directions.



4.1.7 Complete graph:

A complete graph is a graph normally undirected with an edge between each pair of vertices.

Example



4.1.8 Incident edge:

The edge vw is said to be incident, if it is incident on the vertices v and w, and vice versa.

4.1.9 Adjacency Relation:

Adjacency relation in digraph is represented by vAw (w is adjacent to v) if and only if vw is in E.

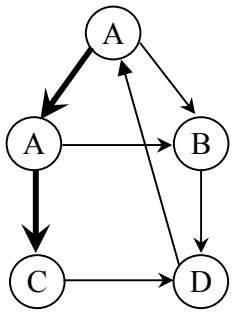
If G is undirected graph, the adjacency relation (A) is symmetric (i.e. wAv if and only if vAw)

4.1.10 Path:

A path from v to w in a graph G = (V,E) is a sequence of edges, v_0v_1, v_1v_2,…..$v_{k-1}v_k$, Where $v=v_0$ and $w=v_k$. The length of the path is k.

Therefore, Length of a path = No.of edges traversed

Example:



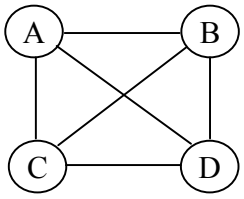
Path length from A to D is 2

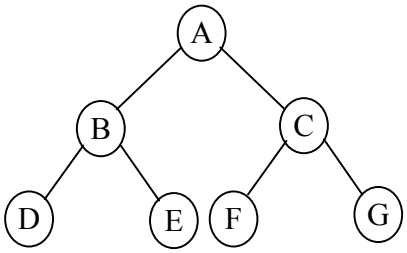
Note: Path length from vertex V to itself is zero.

4.1.11 Connected graph:

An undirected graph is connected if and only if, for each pair of vertices v and w, there is a path from v to w.

Example:

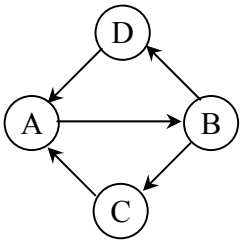




4.1.12 Strongly Connected Graph:

A directed graph is strongly connected if and only if, for each pair of vertices v and w, there is a path from v to w. i.e, **strong connectivity** means travelling the one-way streets in their correct direction from anywhere to anywhere.

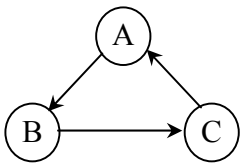
Example:



4.1.13 Cycle:

Cycle in a directed graph is a simple cycle in which no vertex is repeated except that the first and last are identical.

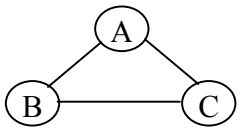
Example:



Cycle = A→ B → C → A

The definition of cycle is same for undirected graph also, but there is a requirement that if any edge appears more than once it always with the same orientation. i.e using the path definition if $v_i = x$ and $v_{i+1} = y$ for $0 \le i < k$, then there cannot be a j such that $v_j = y$ and $v_{j+1} = x$.

Example:



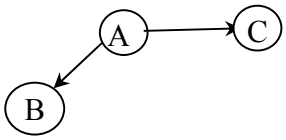
Cycle = ABCA

4.1.14 Acyclic graph:

A graph is acyclic graph if it has no cycles.

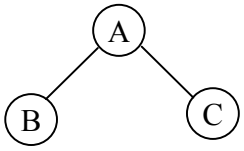
Example:

(a)



DAG – Directed acyclic graph (no specific cycle)

(b)



Undirected acyclic graph (or) undirected tree.

4.1.15 Connected component:

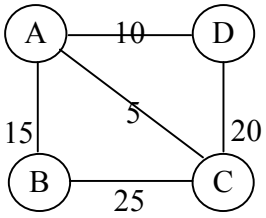
A connected component of an undirected graph G is a maximal connected subgraph of G.

Maximal: A graph is said to be maximal within some collection of graphs if it is not a proper subgraph of any graph in that collection. The word "collection" means that all connected subgraphs of G.

4.1.16 Weighted graph:

A weighted graph is a triple(V,E,W) where (V,E) is a graph (directed or undirected) and W is a function from E into R. For an edge E, W(E) is called the weight of E.

Example:



A—D = 10, represents weight of the edge.

4.1.17 Degree

The number of edges incident on a vertex determines its degree. The degree of the vertex V is written as degree (V).

The *in degree* of the vertex V, is the number of edges entering into the vertex V.

Similarly the *out degree* of the vertex V is the number of edges exiting from that vertex V.

4.2 GRAPH REPRESENTATIONS AND DATA STRUCTURE:

Let G=(V,E) be a graph with n=|V|, number of vertices, m=|E|, number of Edges and V={v_1,v_2,v_3…….v_n}There are several representations for graphs are possible. We can discuss only three most commonly used. They are

1. Adjacency matrices
2. Adjacency lists
3. Adjacency multilists
4. Incidence matrix

The choice of a particular representation will depend upon the application we expect on graph

4.2.1 Adjacency Matrix Representation:

There are mainly two components in a graph (i.e) nodes and edges. Adjacent matrix is the matrix, which keeps the information of adjacent nodes. This matrix keeps the information that whether a node is adjacent to any other node or not.

Let G = (V, E) be a graph with n vertices, n ≥ 1. The adjacency matrix of G is a 2 – dimensional n * n or array[n][n], where the first subscript will be row and second subscript will be column of that matrix, say A, with the property that

A (i, j) = 1 if there is an edge from node V_i, to V_j and is in E (G)

= 0 if there is no edge from node V_i, to V_j

Hence all the entries of this matrix will be either 1 or 0. The edge weight is 1 for an undirected graph.

A graph containing n vertices can be represented by a matrix with n rows and n columns. The matrix is formed by storing the edge weight in its i^{th} row and j^{th} column of the matrix, if there exists an edge between i^{th} and j^{th} vertex of the graph and a '0' if there is no edge between i^{th} and j^{th} vertex of the graph, such a matrix is referred as an adjacency matrix.

AdjMat[i][j] = Weight of the edge, if there is a path from vertex V_i to V_j

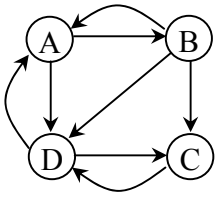
AdjMat[i][j] = 0, otherwise

Now we are going to form a adjacency matrix for,

1. Adjacency matrix for Directed graph
2. Adjacency matrix for Undirected graph
3. Adjacency matrix for Weighted graph

1. Adjacency matrix for Directed graph

Let us consider the graph. The corresponding adjacency matrix for this graph will be



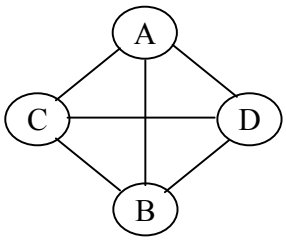
	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	0	0	1
D	1	0	1	0

Adjacency Matrix

The adjacency matrix is maintained in the array arr[4][4]. Here the entry of matrix arr[0][1] = 1, which represents there is an edge in the graph from node A to B. Similarly arr[2][0]= 0, which represents there is no edge from node C to node A.

2. Adjacency matrix for Undirected graph:

Let us consider an undirected graph and the corresponding adjacency matrix for this graph will be,

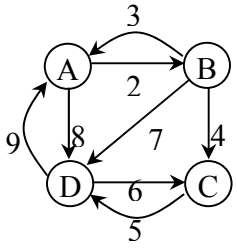


	A	B	C	D
A	0	1	1	1
B	1	0	1	1
C	1	1	0	1
D	1	0	1	0

Adjacency Matrix

The adjacent matrix for an undirected graph will be a symmetric matrix. This implies that for every I and j, A[i][j] = A[j][i] in an undirected graph.

3. Adjacency matrix for Weighted graph:



	A	B	C	D
A	0	2	∞	8
B	3	0	4	7
C	∞	∞	0	5
D	9	∞	6	0

Adjacency Matrix

The space needed to represent a graph using its adjacency matrix is n^2 bits.

Disadvantages:

1. Takes $O(n^2)$ space to represents the graph
2. It takes $O(n^2)$ time to solve the most of the problems.

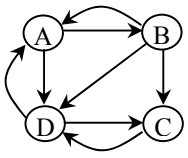
4.2.2 Adjacency List Representation:

- A graph containing n vertices and m edges can be represented using a linked list, referred to as adjacency list. The number of vertices in the graph forms a singly linked list.
- In this representation the n rows of the adjacency matrix are represented as n linked lists.
- Each vertex have a separate linked list, with nodes equal to the number of edges connected from the corresponding vertex. The nodes in list i represent the vertices that are adjacent from vertex i.
- Each node has atleast two fields: VERTEX and LINK. VERTEX fields contain the indices of the vertices adjacent to vertex i.

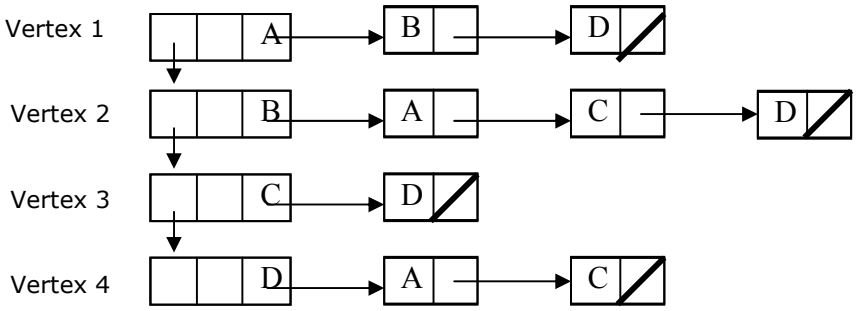
Advantages:

- Eventhough adjacency list is a difficult way of representing graphs, a graph with large number of nodes will use small amount of memory.
- In adjacency list representation of graph, we will maintain two lists. First list will keep track of all the nodes in the graph and second list will maintain a list of adjacent nodes for each node.

1. Adjacency lists for Directed graph:

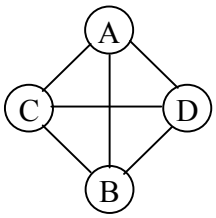


Adjacency list for the above graph is as follows:

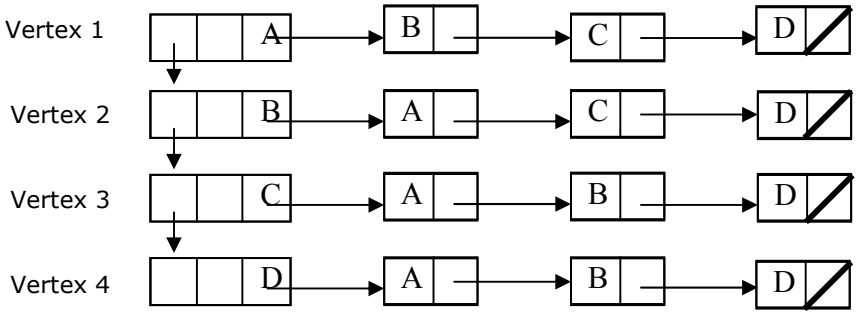


2. Adjacency lists for Undirected graph:

Let us consider the following undirected graph,

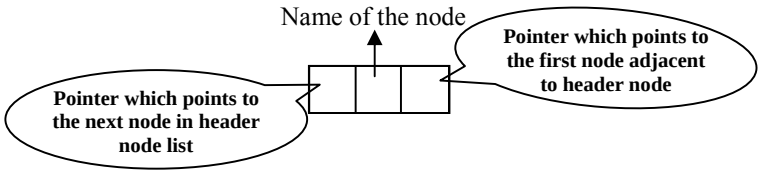


The adjacency list for the above graph is as follows:



In this representation each list had a head node. The head nodes are sequential providing easy random access to the adjacency list for any particular vertex. For an undirected graph with n vertices and m edges this representation requires n head nodes and 2m list nodes. The storage required is then log n + log m for the list nodes and O (log n) for head nodes because 2 fields in each node.

The structure of header node is:



The total number of edges in G may, therefore can be determined by counting the number of nodes on its adjacency list. In order to determine in-degree of a vertex easily, we can maintain one more list called inverse-adjacency lists for each vertex. Each list will contain a node for each vertex adjacent to the vertex it represents.

Disadvantages:

It takes O(n) time to determine whether there is an arc from vertex i to vertex j. Since there can be O(n) vertices on the adjacency list for vertex i.

4.2.3 Adjacency Multilists Representation:

In adjacency list representation of an undirected graph each edge (V_i, V_j) is represent by two entries, one on the list for V_i and other on the List for V_j. Using Multilists each edge will be in exactly one node, but this node will be in two lists. i.e the adjacency lists for each of the two nodes it is incident to the node structural now becomes

M V_1 V_2 Link1 for V_1 Link2 for V_2

Here M is a one bit mark field that may be used to indicate whether or not the edge has been examined.

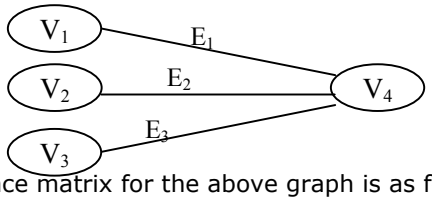
4.2.4 Incidence Matrix Representation:

A graph containing n vertices and m edges can be represented by a matrix with n rows and m columns. The matrix is formed by storing: '1' in its i^{th} row and j^{th} column corresponding to the matrix, if there exists as a i^{th} vertex, connected to one end of the j^{th} edge and a '0' if there is no j^{th} vertex connected to an end of the j^{th} edge of the graph, such a matrix is referred as an incidence matrix.

IncMat[i][j] = 1, if there is an edge E_j from V_i.

IncMat[i][j] = 0, otherwise

Example 1: Let us consider the following undirected graph,



The incidence matrix for the above graph is as follows:

	E_1	E_2	E_3
V_1	1	0	0
V_2	0	1	0
V_3	0	0	1
V_4	1	1	1

Disadvantage of Incidence matrix:

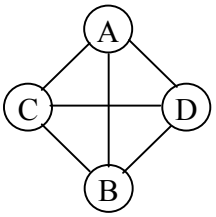
1. Eventhough incidence matrix is an easy way of representing graphs, a graph with considerable number of nodes could use a large amount of memory.
2. Most of the incidence matrices will be sparse matrices only(a matrix with lot of zeros in it).

4.3 APPLICATIONS OF GRAPHS:

The problems using graph representation are useful in many fields for different operations. Some examples are:

4.3.1 Airline route map-undirected graph

The vertices are cities, a line connects two cities if and only if there is a nonstop flight between them in both directions.



A graph between four cities = {A, B, C, D}

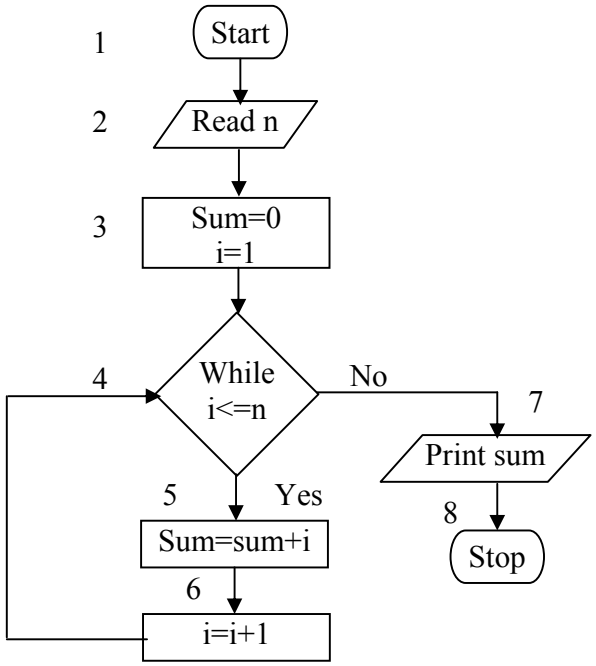
4.3.2 An electrical circuit:

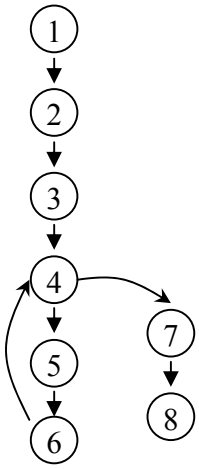
The vertices could be diodes, transistors, capacitors, Switches and so on. Two vertices are connected by a line if there is a wire connecting them.

4.3.3 Flowcharts – Directed graph

The vertices are the flowchart boxes; the connecting arrows are the flowchart arrows.

Flowchart: Directed graph:

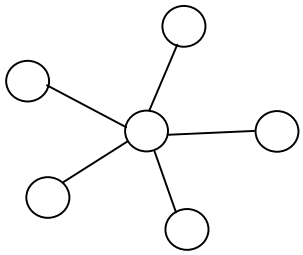




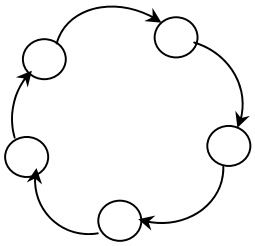
4.3.4 Computer Networks:

The vertices are computers. The lines (undirected graph) or arrows (directed graph) are the communication links.

Example:



a) A star network
(undirected graph)



b) A ring network
(directed graph)

4.4 TOPOLOGICAL SORT

A **topological sort** is a linear ordering of vertices in a Directed Acyclic Graph(DAG) such that if there is a path from V_i to V_j, then V_j appears after V_i in the linear ordering.

Topological ordering is not possible, if the graph has a cycle, since for two vertices v and w on the cycle, v precedes w and w precedes v.

4.4.1 Simple Topological sort:

The Strategy:

(i) Find any vertex with no incoming edges (i.e) indegree=0. If such vertex found, print the vertex and remove it along with its edges from the graph.

(ii) Repeat step (i) for the rest of the graph.

The steps to perform simple topological sort is:

(i) Assuming that the Indegree array is initialized and the graph is read into an adjacency list.

(ii) The function FindNewVerrtexOfIndegreeZero scans the indegree array, to find a vertex with indegree 0, that has not already been assigned a topological number.

a). It returns NotVertex if no such vertex exists, that indicates that the graph has a cycle.

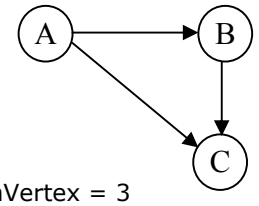
b). If vertex (v) is returned, the topological number is assigned to v, then the indegree of vertices(w) adjacent to vertex (v) are decremented.

(iii) Repeat step 2 for the rest of the graph.

Algorithm for Simple Topological sort:

```
void Topsort(Graph G)
{
    int counter;
    vertex V, W;
    for(counter = 0; counter <NumVertex; counter++)
    {
        V=FindNewVerrtexOfIndegreeZero();
        if(V == NotAVertex)
        {
            Error("Graph has a Cycle");
            Break;
        }
        TopNum[V] = counter;
        for each W adjacent to V
            Indegree[W]--;
    }
}
```

Example:



- NumVertex = 3
- Indegree is initialized in an array
- Graph is stored in an adjacency list.

Step 1: Find a vertex with Indegree 0 (i.e) a vertex with no incoming edges. Therefore,

V= A is returned

Step 2: Topological Number is assigned to vertex A.

TopNum[A] = 0

The indegree adjacent vertices of V are decremented, (i.e) B, C

Now Indegree[B] = 0

Indegree[c] = 1

Step 3: Step (1) is repeated.

Find a vertex with indegree 0 (i.e) a vertex with no incoming edges. Therefore,

V= B is returned

Step 4: Step (2) is repeated

Topological Number is assigned to vertex B.

TopNum[B] = 1

The indegree adjacent vertices of V are decremented, (i.e) C

Now Indegree[c] = 0

Step 5: Step (1) is repeated.

Find a vertex with indegree 0 (i.e) a vertex with no incoming edges. Therefore,

V= C returned

Step 6 Step (2) is repeated

Topological Number is assigned to vertex C

TopNum[c]= 2

Step 7: Loop variable counter !< NumVertex so the iteration process for finding the vertex was terminated.

Step 8: The vertices are printed as per the topological number ordering.

i.e, A, B, C → hence the topological order is derived for the given DAG.

Analysis

In this algorithm FindNewVerrtexOfIndegreeZero is a simple sequential scan of Indegree array, each call takes (O|V|) time. Since there are |V| such calls, the running time of the algorithm is ($O|V|^2$).

Drawback of this algorithm:

- If the graph is sparse, only a few vertices have their Indegree to be updated during iteration. But this algorithm carries a sequential search of all the vertices.

To Overcome This:

A queue is to be maintained for vertices with Indegree 0 and not yet processed.

4.4.2 To implement the topological sort with queue representation, perform the following steps.

Step 1 : - Find the indegree for every vertex.

Step 2 : - Place the vertices whose indegree is `0' on the empty queue.

Step 3 : - Dequeue the vertex V and decrement the indegree's of all its adjacent vertices.

Step 4 : - Enqueue the vertex on the queue, if its indegree falls to zero.

Step 5 : - Repeat from step 3 until the queue becomes empty.

Step 6 : - The topological ordering is the order in which the vertices dequeued.

Routine to perform Topological Sort using queue representation

/* Assume that the graph is read into an adjacency matrix and that the indegrees are computed for every vertices and placed in an array (i.e. Indegree [ ]) */

```
void Topsort (Graph G)
{
    Queue Q ;
    int counter = 0;
    Vertex V, W ;
    Q = CreateQueue (NumVertex);
    Makeempty (Q);
    for each vertex V
    if (indegree [V] = = 0)
    Enqueue (V, Q);
    while (! IsEmpty (Q))
    {
        V = Dequeue (Q);
        TopNum [V] = + + counter;
        for each W adjacent to V
        if (--Indegree [W] = = 0)
        Enqueue (W, Q);
    }
    if (counter ! = NumVertex)
    Error (" Graph has a cycle");
    DisposeQueue (Q); /* Free the Memory */
}
```

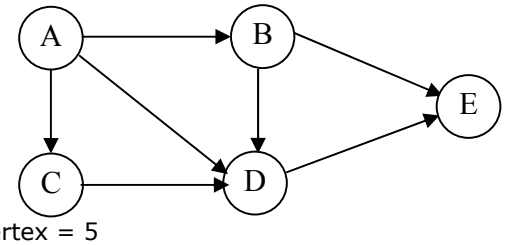
Note :

Enqueue (V, Q) implies to insert a vertex V into the queue Q.

Dequeue (Q) implies to delete a vertex from the queue Q.

TopNum [V] indicates an array to place the topological numbering.

Illustration with an example:



- NumVertex = 5
- Indegree is intialised in an array
- Graph is stored as an adjacency list

VERTEX	Indegree before Dequeue				
	1	**2**	**3**	**4**	**5**
A	**0**	**0**	**0**	**0**	**0**
B	**1**	**0**	**0**	**0**	**0**
C	**1**	**0**	**0**	**0**	**0**
D	**3**	**2**	**1**	**0**	**0**
E	**2**	**2**	**1**	**1**	**0**
Enqueue	**A**	**B,C**		**D**	**E**
Dequeue	**A**	**B**	**C**	**D**	**E**

Step 1

Number of 1's present in each column of adjacency matrix represents the indegree of the corresponding vertex.

From above figure Indegree [A] = 0, Indegree [B] = 1, Indegree [C] = 1, Indegree [D] = 3,

Indegree [E]= 2

Step 2

Enqueue the vertex, whose indegree is `0'

Vertex `A' is 0, so place it on the queue and then dequeued.

Step 3

Dequeue the vertex `A' from the queue and decrement the indegree's of its adjacent vertex `B' & `C'

Hence, Indegree [B] = 0 & Indegree [C] = 0

Now, Enqueue the vertex `B' AND 'C' as its indegree becomes zero. Then dequeue the vertex 'B'.

Step 4

Dequeue the vertex `B' from Q and decrement the indegree's of its adjacent vertex `C' and `D'.

Hence, Indegree [C] = 0 & Indegree [D] = 1

Now, Enqueue the vertex `C' as its indegree falls to zero.

Step 5

Dequeue the vertex `C' from Q and decrement the indegree's of its adjacent vertex `D'.

Hence, Indegree [D] = 0

Now, Enqueue the vertex `D' as its indegree falls to zero.

Step 6

Dequeue the vertex `D'

Step 7

As the queue becomes empty, topological ordering is performed, which is nothing but, the order in which the vertices are dequeued.

Analysis

The running time of this algorithm is O(|E| + |V|). where E represents the Edges & V represents the vertices of the graph.

4.4.3 Implementation of Topological sorting technique:

(Refer the lab program manual and write only the corresponding functions)

4.5 GRAPH TRAVERSALS

Graph traversal is defined as visiting all the vertices of the graph G = (V, E), where V is the set of vertices and E is the set of edges.

Two different types of traversal strategies are:

1. Depth First Search (DFS)
2. Breadth First Search (BFS)

4.5.1 Depth First Search (DFS) :

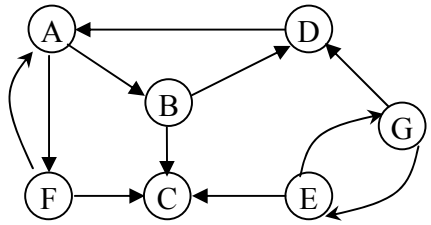
- DFS is otherwise known as Depth first Traversal(DFT), it is kind of tree traversal.
- DFS technique is as follows:
  - Starting vertex may be determined by the problem or chosen arbitrarily. Visit the start vertex v, next an unvisited vertex w adjacent to v is selected and a DFS from w is initiated. When a vertex u is reached such that all the adjacent vertices have been visited, we back up to the last vertex visited which has an unvisited vertex w adjacent to it and initiate a DFS from w. The search terminates when no unvisited vertex can be reached from any of the initiated vertices.
  - If path exists from one node to another node walk across the edge – *exploring the edge*.
  - If path does not exist from one specific node to any other node, return to the previous node where we have been before – *backtracking*.
- The theme of DFS is to explore if possible, otherwise backtrack.

Algorithm:

```
Procedure DFS(u)
    VISITED (u) ← 1
     for each vertex w adjacent to v do
     if VISITED ( w ) = 0 then call DFS ( w )
     end
end DFS
```

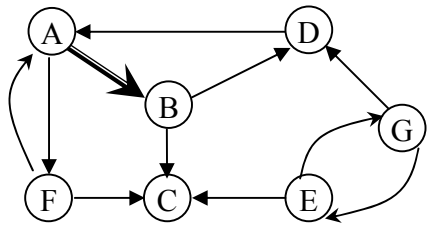
Example:

a)



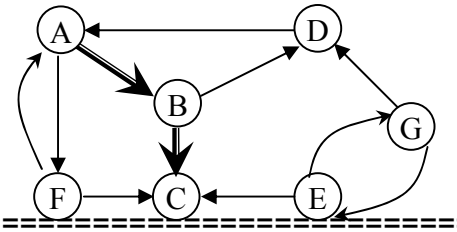
- A directed graph G=(V,E) is given, where V={ A, B, C, D, E, F, G }
- For simplicity, assume the start vertex is A and exploration is done in alphabetical order.

b)



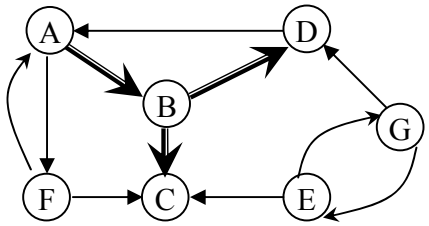
- From start vertex A explores to B, now AB is *explored edge.*

c)



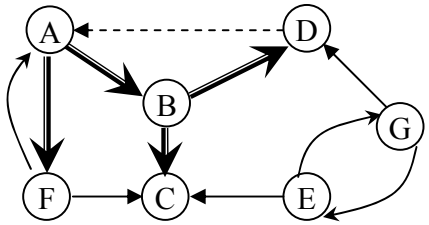
- From vertex B either C or D to be explored, but C is explored as per the alphabetical order.
- There is nowhere to explore from C, therefore C is a *dead end*.

d)



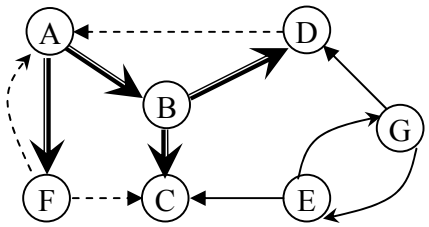
- Since C is a dead end, *backtrack* to B, from there explore to D

e)



- From D it is possible to explore A, this would complete a cycle, but trees should not have cycles.
- Again backtrack to B, from there backtrack to A, explore the path to F.

f)



- From F it is possible to traverse either A or C, but both the nodes are discovered nodes. So F is also a *dead end*.
- From the above diagram it is possible to say that G and E are never traversed.

4.5.2 Breadth First Search:

Starting at vertex v and marking it as visited. BFS performs simultaneous explorations starting from a common point and spreading out independently.

Breadth firs search differs from depth first search in that all unvisited vertices adjacent to v are visited next. Then unvisited vertices adjacent to these vertices are visited and so on. A breadth first search beginning at vertex v1 of the graph in above figure would first visit root node and then one by one in the same level and passes to other level.

BFS Technique steps:

1. Put the ending node (the root node) in the queue.
2. Pull a node from the beginning of the queue and examine it.
   - If the searched element is found in this node, quit the search and return a result.
   - Otherwise push all the (so-far-unexamined) successors (the direct child nodes) of this node into the end of the queue, if there are any.
3. If the queue is empty, every node on the graph has been examined -- quit the search and return "not found".
4. Repeat from Step 2.

Procedure BFS

//Breadth first search of G is carried out beginning at vertex v. All the vertices visited are marked as VISITED(i) = 1.The graph G and array VISITED are global and VISITED is initialized to zero //

VISITED (v) ← 1

Initialize Q to be empty // Q is queue //

```
                    Loop
                        for all vertices w adjacent to v do
                            if VISITED(w) = 0          // add w to queue //
                                then [ call ADDQ(w, Q); VISITED(w) ← 1 ]
                             end                                  // mark w as VISITED//
                              if Q is empty then return
                              call DELETEQ ( v, Q )
                         forever
            end BFS
```

Implementation:

```
void BFS(VLink G[], int v)
{
    int w; VISIT(v);                /*visit vertex v*/
    visited[v] = 1;             /*mark v as visited : 1 */
    ADDQ(Q,v);
    while(!QMPTYQ(Q))
    {
        v = DELQ(Q);              /*Dequeue v*/
        w = FIRSTADJ(G,v);        /*Find first neighbor, return -1 if no neighbor*/
        while(w != -1)
        {
            if(visited[w] == 0)
            {
                VISIT(w);             /*visit vertex v*/
                ADDQ(Q,w);        /*Enqueue current visited vertex w*/
                visited[w] = 1;       /*mark w as visited*/
            }
           W = NEXTADJ(G,v);   /*Find next neighbor, return -1 if no neighbor*/
        }
    }
}
```

Main Algorithm of apply Breadth-first search to graph G=(V,E) :

```
void TRAVEL_BFS(VLink G[], int visited[], int n)
{
    int i;
    for(i = 0; i < n; i ++)
    {
        visited[i] = 0;          /* Mark initial value as 0 */
    }
    for(i = 0; i < n; i ++)
        if(visited[i] == 0)
```

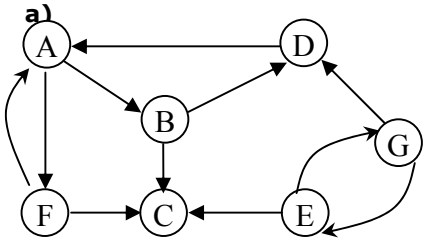
```
            BFS(G,i);
        }
```

Applications of Graph Traversal: -

1. Finding the components of a graph and
2. Finding a spanning tree of a connected graph
3. Finding the shortest path between two nodes *u* and *v* (in an unweighted graph)

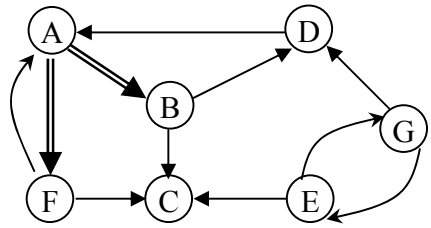
Example:

a)



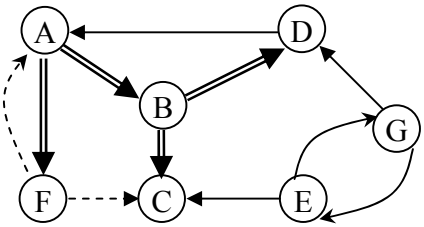
- A directed graph G=(V,E) is given, where V={ A, B, C, D, E, F, G }
- Assume the start vertex is A.
- Explore all paths from vertex A

b)



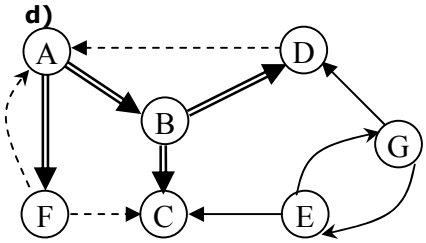
- From A the *explored edges* are AF, AB

c)



- Explore all paths from vertex B and F.
- From B the explored edges are BD, BC.
- From F the explored edges are FA, FC.
- *The dashed lines shows the edges that were explored but went to vertices that were previously discovered.(i.e FA, FC)*

d)



- From D the explored edge is DA.
- But A is already exist in the discovered vertices list. So we will say that the edge DA is *checked* rather than *explored*.

** There is no backtracking in BFS and E and G are unreachable.

4.5.3 DFS vs BFS

Depth first search	Breadth first search
1. Backtracking is possible from a dead end	1. Backtracking is not possible.
2. Vertices from which exploration is incomplete are processed in a LIFO order.	2. The vertices to be explored are organized as a FIFO queue.
3. Search is done in one particular direction at the time.	3. Search is done parallely in all possible direction.

4.6 SHORTEST PATH ALGORITHM

The Shortest path algorithm determines the minimum cost of the path from source to every other vertex. The cost of the path V_1, V_2, --V_N is

$$\sum_{i=1}^{n-1} C_{i,i+1} .$$

This is referred as weighted path length. The unweighted path length is the number of the edges on the path, namely N - 1.

Two types of shortest path problems, exist namely,

1. **Single source shortest path problem**
   The single source shortest path algorithm finds the minimum cost from single source vertex to all other vertices. For a given graph G=(V,E), the shortest path is calculated from a distinguished vertex S to every other vertex in G. Dijkstra's algorithm is used to solve this problem which follows the greedy technique.
2. **All pairs shortest path problem**
   All pairs shortest path problem finds the shortest path for a given graph G=(V,E) of every pair of vertices in G. To solve this problem dynamic programming technique known as floyd's algorithm is used.

These algorithms are applicable to both directed and undirected weighted graphs provided that they do not contain a cycle of negative length. Here we will single source shortest path problem for unweighted and weighted graph.

4.6.1 Single Source Shortest Path

Given an input graph G = (V, E) and a distinguished vertex S, find the shortest path from S to every other vertex in G. This problem could be applied to both weighted and unweighted graph.

4.6.1.1 Unweighted Shortest Path

In unweighted shortest path all the edges are assigned a weight of "1" for each vertex. The following three pieces of information is maintained.

known

Specifies whether the vertex is processed or not. It is set to `1' after it is processed, otherwise `0'. Initially all vertices are marked unknown. (i.e) `0'.

d_v

Specifies the distance from the source `s', initially all vertices are unreachable except for 's', whose path length is `0'.

P_v

Specifies the bookkeeping variable which will allow us to print. The actual path (i.e) the vertex which makes the changes in dv.

To implement the unweighted shortest path, perform the following steps :

Step 1 : - Create a queue, where the size of the queue is equivalent to the number of vertices in the graph.

Step 2 : - Empty the queue.

Step 3 : - Arbitrarily assign the source node as `S' and Enqueue `S'.

Step 4 : - Dequeue the vertex `S' from queue and assign the value of that vertex to be known and then find its adjacency vertices. Also update the three parameters known, d_v and P_v.

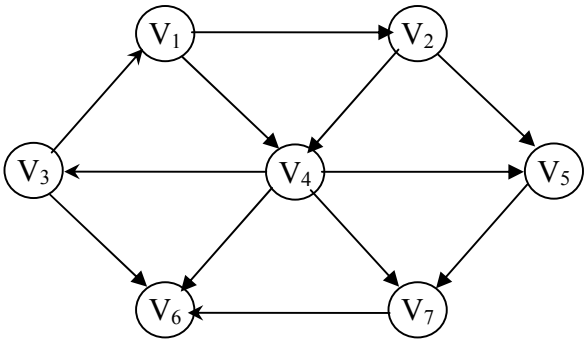
Step 5 :- If the distance of the adjacent vertices is equal to infinity then change the distance of that vertex as the distance of its source vertex increment by `1' and Enqueue the vertex.

Step 6 :- Repeat from step 2, until the queue becomes empty.

ROUTINE FOR UNWEIGHTED SHORTEST PATH

```
void Unweighted (Table T)
{
        Queue Q;
        Vertex V, W ;
        Q = CreateQueue (NumVertex);
        MakeEmpty (Q);
        /* Enqueue the start vertex s */
        Enqueue (s, Q);
        while (! IsEmpty (Q))
        {
                V = Dequeue (Q);
                V = Dequeue (Q);
                T[V]. Known = True; /* Not needed anymore*/
                for each W adjacent to V
                if (T[W]. Dist = = INFINITY)
                {
                        T[W] . Dist = T[V] . Dist + 1 ;
                        T[W] . path = V;
                        Enqueue (W, Q);
                }
        }
        DisposeQueue (Q) ; /* Free the memory */
}
```

Illustrations with an example 1:



Vertex	Initial State			V_3 Dequeued		
	Known	d_v	P_v	Known	d_v	P_v
V_1	0	∞	0	0	1	V_3
V_2	0	∞	0	0	∞	0
V_3	0	0	0	1	0	0
V_4	0	∞	0	0	∞	0
V_5	0	∞	0	0	∞	0
V_6	0	∞	0	0	1	V_3
V_7	0	∞	0	0	∞	0
Queue		V_3			V_1, V_6	

Vertex	V_1 Dequeued			V_6 Dequeued		
	Known	d_v	P_v	Known	d_v	P_v
V_1	1	1	V_3	1	1	V_3
V_2	0	2	V_1	0	2	V_1
V_3	1	0	0	1	0	0
V_4	0	2	V_2	0	2	V_1
V_5	0	∞	0	0	∞	0
V_6	0	2	V_3	1	1	V_3
V_7	0	∞	0	0	∞	0
Queue		V_6, V_2, V_4			V_2, V_4	

Vertex	V_2 Dequeued			V_4 Dequeued		
	Known	d_v	P_v	Known	d_v	P_v
V_1	1	1	V_3	1	1	V_3
V_2	1	2	V_1	1	2	V_1
V_3	1	0	0	1	0	0
V_4	0	2	V_1	1	2	V_1
V_5	0	3	V_2	0	3	V_2
V_6	1	1	V_3	1	1	V_3
V_7	0	∞	0	0	3	V_4
Queue		V_4, V_5			V_5, V_7	

Vertex	V_5 Dequeued			V_7 Dequeued		
	Known	d_v	P_v	Known	d_v	P_v
V_1	1	1	V_3	1	1	V_3
V_2	1	2	V_1	1	2	V_1
V_3	1	0	0	1	0	0
V_4	1	2	V_1	1	2	V_1
V_5	1	3	V_2	1	3	V_2
V_6	1	1	V_3	1	1	V_3

V_7	0	3	V_4	1	3	V_4
Queue		V_7			Empty	

In general, when the vertex 'V' is dequeued and the distance of its adjacency vertex 'W' is infinitive then distance and path of 'W' is calculated as follows:

T[W].Dist = T[V].Dist + 1

T[W].path = V

When V_3 is dequeued, *known* is set to '1' for that vertex and the distance of its adjacent vertices V_1 and V_6 are updated if INFINITY. Path length of V_1 is calculated as;

$T[V_1].Dist = T[V_3].Dist + 1 = 0 + 1 = 1$

And its actual path is also calculated as,

$T[V_1].path = V_3$

Similarly,

$T[V_6].Dist = T[V_3].Dist + 1 = 0 + 1 = 1$

$T[V_6].path = V_3$

Similarly, when the other vertices are dequeued, the table is updated as shown above. The shortest distance from the source vertex V_3 to all other vertex is listed below;

$V_3 \rightarrow V_1$ is 1

$V_3 \rightarrow V_2$ is 2

$V_3 \rightarrow V_4$ is 2

$V_3 \rightarrow V_5$ is 3

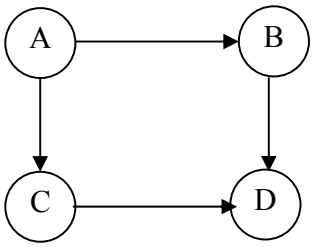
$V_3 \rightarrow V_6$ is 1

$V_3 \rightarrow V_7$ is 3

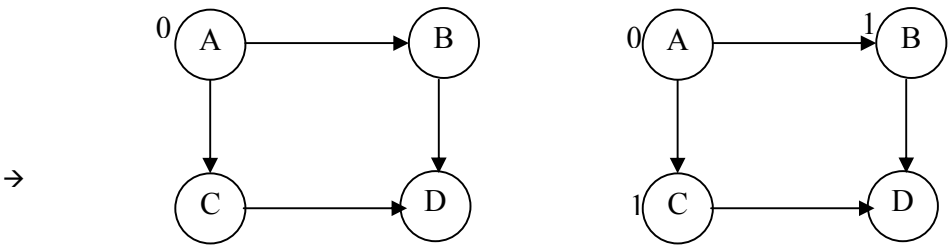
Analysis: The running time of this algorithm is O(|E|+|V|) if adjacency list is used to store and rread the graph.

Illustrations with an example 2:

Find the shortest path for the following graph with 'A' as source vertex.

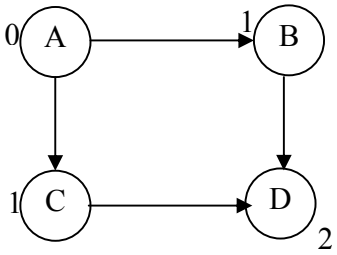


Solution: Source vertex `A' is initially assigned a path length `0'.



Vertex	Initial State			A Dequeued		
	Known	d_v	P_v	Known	d_v	P_v
A	0	0	0	1	0	0
B	0	∞	0	0	1	A
C	0	∞	0	0	1	A
D	0	∞	0	0	∞	0
Queue		A			B, C	

After finding all vertices whose path length from 'A' is '1'.



Vertex	B Dequeued			C Dequeued			D Dequeued		
	Known	d_v	P_v	Known	d_v	P_v	Known	d_v	P_v
A	1	0	0	1	0	0	1	0	0
B	1	1	A	1	1	A	1	1	A
C	0	1	A	1	1	A	1	1	A
D	0	2	B	0	2	B	1	2	B
Queue		C, D			D			Empty	

The shortest distance from the source vertex A to all other vertex is listed below;

A → B is 1

A → C is 1

A → D is 2

4.6.1.2 Weighted Shortest Paths - Dijkstra's Algorithm

The general method to solve the single source shortest path problem is known as Dijkstra's algorithm. This is applied to the weighted graph G.

Dijkstra's algorithm is the prime example of Greedy technique, which generally solve a problem in stages by doing what appears to be the best thing at each stage. This algorithm proceeds in stages, just like the unweighted shortest path algorithm.

At each stage, it selects a vertex v, which has the smallest d_v among all the unknown vertices, and declares that as the shortest path from S to V and mark it to be known. We should set $d_w = d_v + C_{vw}$, if the new value for d_w would be an improvement.

Algorithm(Informal)

1. Create a table with known, P_v, d_v parameters, where the size of the table equivalent to the number of vertices (0 to N-1).

   known

   Specifies whether the vertex is processed or not. It is set to `1' after it is processed, otherwise `0'. Initially all vertices are marked unknown. (i.e) `0'.

d_v

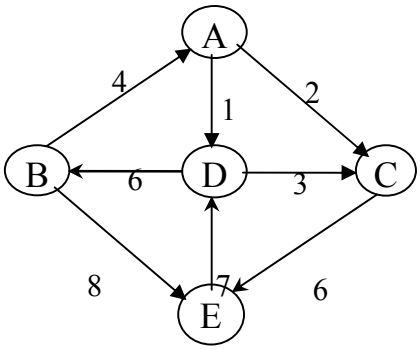
Specifies the distance from the source `s', initially all vertices are unreachable except for 's', whose path length is `0'.

P_v

Specifies the bookkeeping variable which will allow us to print. The actual path (i.e) the vertex which makes the changes in dv.

2. Read graph from the adjacency list representation.
3. Select a vertex V which has smallest dv among all unknown vertices and sets the shortest path from S to V as known.
4. The adjacent vertices W to V is located and dw is set as $d_w = d_v + C_{vw}$, if the new sum is less than the existing dw. That is, in every stage dw is updated only if there is an improvement in the path distance.
5. Repeat step 3 and 4, until all vertices are classified under known.

Example 1 :



The directed graph G=(V, E, W)

Initial configuration of the table

Vertex	Known	d_v	P_v	Comments
A	0	0	0	
B	0	∞	0	A is chosen as source vertex, since the edge AD has minimum value.
C	0	∞	0	
D	0	∞	0	
E	0	∞	0	

After A is declared known

Vertex	Known	d_v	P_v	Comments
A	1	0	0	
B	0	∞	0	The vertices adjacent to A is C and D. Therefore, d_v, P_v of C and D is updated.
C	0	2	A	
D	0	1	A	
E	0	∞	0	

After D is declared known

Vertex	Known	d_v	P_v	Comments
A	1	0	0	The vertices adjacent to D is B and C.
B	0	6	D	The vertex B, through D is included.

C	0	2	A	The d_v, P_v values of vertex C remains
D	1	1	A	same, since the d_v through A is less
E	0	∞	0	than d_v through D.

After C is declared known

Vertex	Known	d_v	P_v	Comments
A	1	0	0	
B	0	6	D	
C	1	2	A	The vertex E is updated, since it is
D	1	1	A	adjacent to C.
E	0	8	C	

After B is declared known

Vertex	Known	d_v	P_v	Comments
A	1	0	0	
B	1	6	D	
C	1	2	A	The vertex B is selected through D.
D	1	1	A	
E	0	8	C	

After E is declared known

Vertex	Known	d_v	P_v	Comments
A	1	0	0	
B	1	6	D	
C	1	2	A	The vertex E is selected through c.
D	1	1	A	
E	1	8	C	

- Initial values of known, d_v, P_v are assigned in a table.
- A is selected as source vertex, since AD=1 (minimum cost function), classified as known.
- From A, the vertices C and D are updated with the given distance passing through A. The vertices C and D are classified under known.
- The vertex B is set to known, with distance 6, passing through D.
- To reach E, the paths with weights

  A – C – E = 8

  A – D – C – E = 10
- The path with low cost is selected, and E is set as known.

ROUTINE FOR DIJKSTRA'S ALGORITHM

```
Void Dijkstra (Graph G, Table T)
{
    int i ;
    vertex V, W;
    Read Graph (G, T) /* Read graph from adjacency list */
    /* Table Initialization */
```

```
        for (i = 0; i < Numvertex; i++)
        {
            T [i]. known = False;
            T [i]. Dist = Infinity;
            T [i]. path = NotA vertex;
        }
        T [start]. dist = 0;
        for ( ; ;)
        {
            V = Smallest unknown distance vertex;
            if (V = = Not A vertex)
            break ;
            T[V]. known = True;
            for each W adjacent to V
            if ( ! T[W]. known)
            {
                T [W]. Dist = Min [T[W]. Dist, T[V]. Dist + C_VW]
                T[W]. path = V;
            }
        }
    }
```

Analysis:

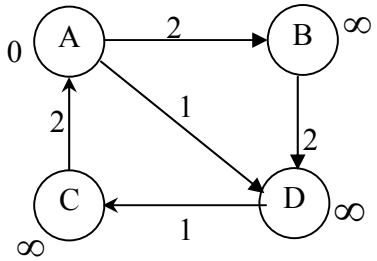
This algorithm works for the weighted graph without negative cost. If any edge has negative cost, the algorithm could produce the wrong answer.

The running time depends on how the table is manipulated. If we use the evident algorithm of scanning down the table to find the minimum d_v, each phase will take O(|V|) time to find the minimum. Thus $O(|V|^2)$ time will be spent for finding the minimum over the course of the algorithm.

The time for updating d_w is constant per update, and there is at most one update per edge for a total of O(|E|). Thus, the total running time is $O(|E|+|V|^2) = O(|V|^2)$.

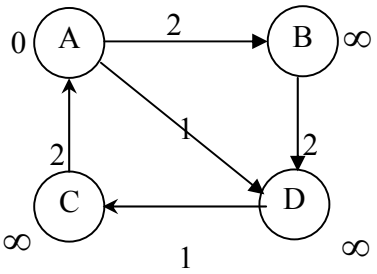
If the graph is dense, with $|E| = E(|V|^2)$, this algorithm is not only simple but essentially optimal, since it runs in time linear in the number of edges. If the graph has negative edge cost, then Dijkstra's algorithm does not work.

Example 2: Find the shortest path for the following weighted graph.



Solution:

Step 1:



Initial Configuration

Vertex	Initial State		
	known	d_v	P_v
A	0	0	0
B	0	∞	0
C	0	∞	0
D	0	∞	0

Step 2:

Vertex `A' is choose as source vertex and is declared as known vertex.

Then the adjacent vertices of `A' is found and its distance are updated as follows :

T [B] .Dist = Min [T[B].Dist, T[A] .Dist + $C_{a,b}$]

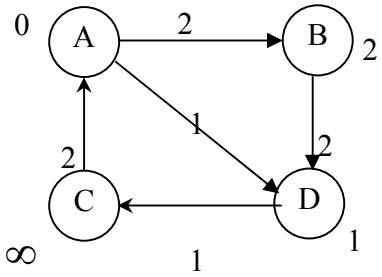
= Min [∞ , 0 + 2]

= 2

T [d] .Dist = Min [T[d].Dist, T[a] .Dist + $C_{a,d}$]

= Min [∞ , 0 + 1]

= 1



After 'A' is declared known

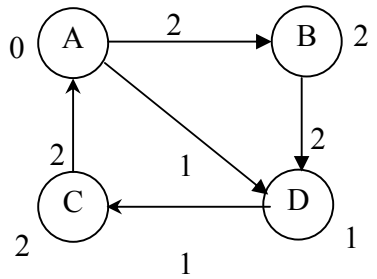
Vertex	known	d_v	P_v
A	1	0	0
B	0	2	A
C	0	∞	0
D	0	1	A

After 'A' is declared Known

Step 3:

Now, select the vertex with minimum distance, which is not known and mark that vertex as visited. Here `D' is the next minimum distance vertex. The adjacent vertex to `D' is `C', therefore, the distance of C is updated a follows, T[C].Dist = Min [T[C].Dist, T[D].Dist + $C_{d,c}$]

= Min [∞ , 1 + 1] = 2



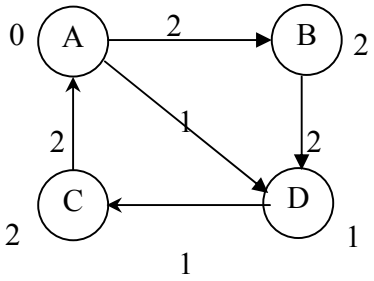
After 'D' is declared known

Vertex	known	d_v	P_v
A	1	0	0
B	0	2	A
C	0	2	D
D	1	1	A

After 'D' is declared Known

Step 4:

The next minimum vertex is 'B' and mark it as visited.



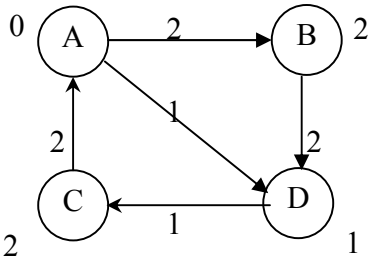
Vertex	known	d_v	P_v
A	1	0	0
B	1	2	A
C	0	2	D
D	1	1	A

After 'B' is declared Known

After 'B' is declared known

Step 5:

Since the adjacent vertex 'D' is already visited, select the next minimum vertex `C' and mark it as visited.



Vertex	known	d_v	P_v
A	1	0	0
B	1	2	A
C	1	2	D
D	1	1	A

After 'C' is declared Known, then the algorithm terminates

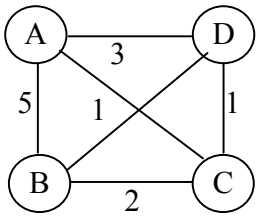
After 'C' is declared known and algorithm terminates

4.7 MINIMUM SPANNING TREE

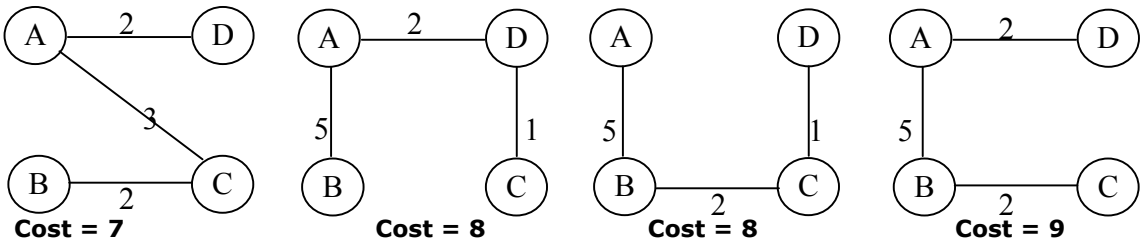
A tree is a connected graph with no cycles. A spanning tree is a subgraph of *G* that has the same set of vertices of *G* and is a tree.

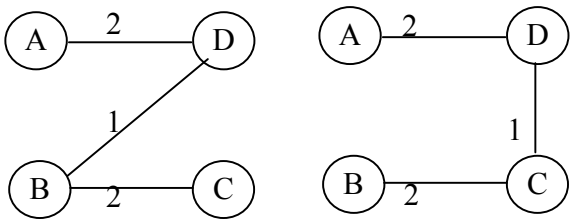
A minimum spanning tree of a weighted connected graph G is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges. The total number of edges in Minimum Spanning Tree (MST) is |V|-1 where V is the number of vertices.

A minimum spanning tree exists if and only if G is connected. For any spanning Tree T, if an edge E that is not in T is added, a cycle is created. The removal of any edge on the cycle reinstates the spanning tree property.



Connected Graph G





Minimum Spanning Tree → Cost = 5 **Cost = 5**

Spanning Tree with minimum cost is considered as Minimum Spanning Tree

Types of Algorithm to find MST:

1. Prim's Algorithm
2. Kruskal's Algorithm

4.7.1 Prim's Algorithm

Prim's algorithm is one of the methods to compute a minimum spanning tree which uses a greedy technique. This algorithm begins with a set U initialised to {1}.

In each stage, one node is picked as the root, and we add an edge, and thus an associated vertex, to the tree, until all the vertices are present in the tree with |V|-1 edges.

At each step, it finds a shortest edge (u,v) such that the cost of (u, v) is the smallest among all edges, where u is in Minimum Spanning Tree and V is not in Minimum Spanning Tree.

Algorithm (Informal)

i) One node is picked as a root node(u) from the given connected graph.

ii) At each stage choose a new vertex v from u, by considering an edge (u,v) with minimum cost among all edges from u, where u is already in the tree and v is not in the tree.

iii) The prim's algorithm table is constructed with three parameters. They are;

- Known – Vertex is added in the tree or not.
- d_v – Weight of the shortest arc connecting v to a known vertex.
- Pv – last vertex which causes a change in d_v.

iv) After selecting the vertex v, update rule is applied for each unknown w adjacent to v. The rule is dw = min(dw, $C_{w,v}$), i.e, if more than one path exist between v to w, then dw is updated with minimum cost.

SKETCH OF PRIM'S ALGORITHM

```
void Prim (Graph G)
{
    MSTTREE T;
    Vertex u, v;
    Set of vertices V;
    Set of tree vertices U;
    T = NULL;
    /* Initialization of Tree begins with the vertex `1' */
    U = {1}
    while (U # V)
```

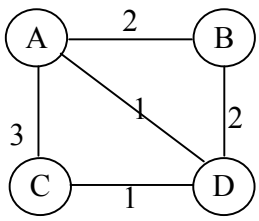
```
        {
            Let (u,v) be a lowest cost such that u is in U and v is in V - U;
            T = T U {(u, v)};
            U = U U {V};
        }
    }
```

ROUTINE FOR PRIMS ALGORITHM

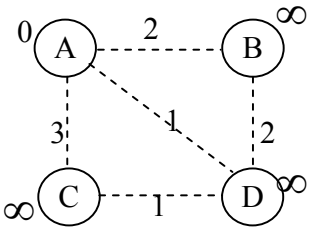
```
    void Prims (Table T)
    {
        vertex V, W;
        /* Table initialization */
        for (i = 0; i < Numvertex ; i++)
        {
            T[i]. known = False;
            T[i]. Dist = Infinity;
            T[i]. path = 0;
        }
        for (; ;)
        {
            Let V be the start vertex with the smallest distance
            T[V]. dist = 0;
            T[V]. known = True;
            for each W adjacent to V
            If (! T[W] . Known)
            {
                T[W].Dist = Min
                (T[W]. Dist, CVW);
                T[W].path = V;
            }
        }
    }
```

Analysis: The implementation of prim's algorithm is virtually identical to that of Dijkstra's algorithm. Prim's algorithm runs on undirected graphs. So when coding it, remember to put every edge in two adjacency lists. The running time is $O(|V|^2)$ without heaps, which is optimal for dense graphs, and $O(|E| \log |V|)$ using binary heaps.

Example 1: For the following graph construct MST using prim's algorithm.



Step 1:



Undirected Graph G

Vertex	Initial State		
	known	d_v	P_v
A	0	0	0
B	0	∞	0
C	0	∞	0
D	0	∞	0

INITIAL CONFIGURATION

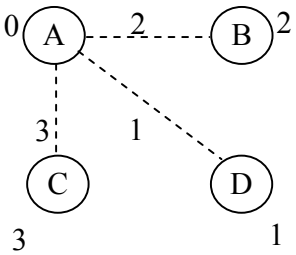
Step 2:

Here, `A' is taken as source vertex and marked as visited. Then the distance of its adjacent vertex is updated as follows :

T[B].dist = Min (T[B].Dist, $C_{a,b}$) T[D].dist = Min [T[D].Dist, $C_{a,b}$] T[C].dist = Min [T[C].Dist, $C_{a,b}$]

= Min (∞, 2) = Min (∞ , 1) = Min (∞ , 3)

= 2 = 1 = 3



Vertex	After 'A' is marked visited		
	known	d_v	P_v
A	1	0	0
B	0	2	A
C	0	3	A
D	0	1	A

Step 3:

Next, vertex `D' with minimum distance is marked as visited and the distance of its unknown adjacent vertex is updated.

T[B].Dist = Min [T[B].Dist, $C_{d,b}$]

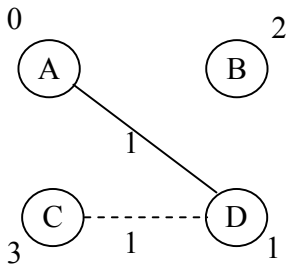
= Min (2, 2)

= 2

T[C].dist = Min [T[C].Dist, $C_{d,c}$]

= Min (3,1)

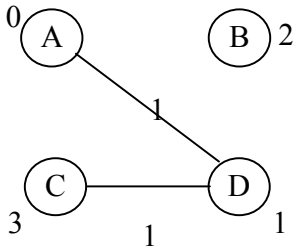
=1



Vertex	After 'D' is marked visited		
	known	d_v	P_v
A	1	0	0
B	0	2	A
C	0	1	D
D	1	1	A

Step 4:

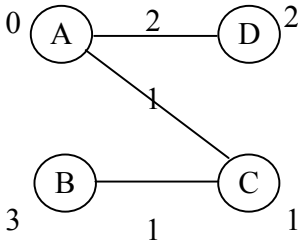
Next, the vertex with minimum cost `C' is marked as visited and the distance of its unknown adjacent vertex is updated.



Vertex	After 'C' is marked visited		
	known	d_v	P_v
A	1	0	0
B	0	2	A
C	1	1	D
D	1	1	A

Step 5:

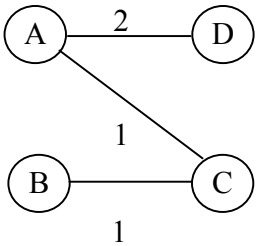
Since, there is no unknown vertex adjacent to `C', there is no updation in the distance. Finally, the vertex `B' which is not visited is marked.



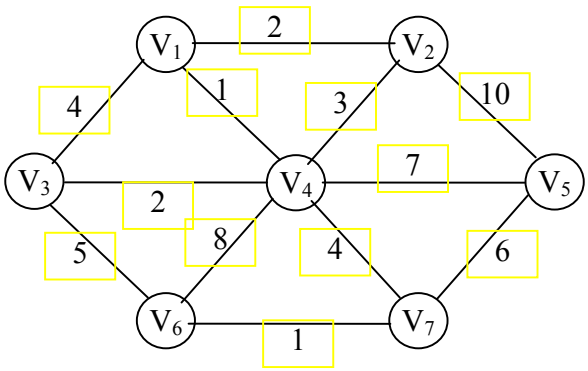
Vertex	After 'B' is marked visited Algorithm terminates.		
	known	d_v	P_v
A	**1**	0	0
B	1	2	A
C	1	1	D
D	1	1	A

The final MST using prim's algorithm is as follows:

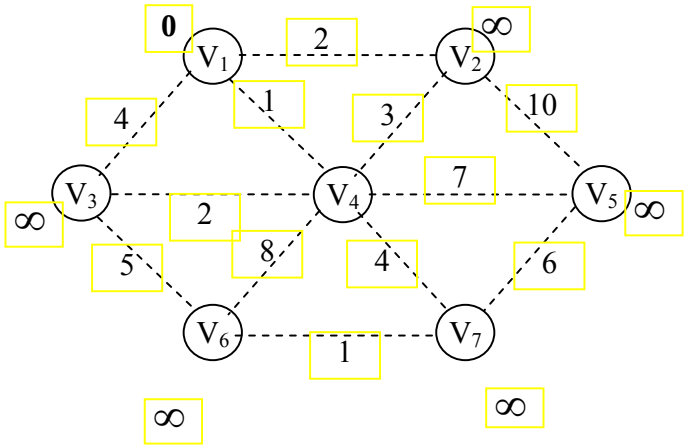
The minimum cost of this spanning Tree is 4 [i.e $C_{a,b} + C_{a,d} + C_{c,d}$]



Example 2: For the following graph construct MST using prim's algorithm.

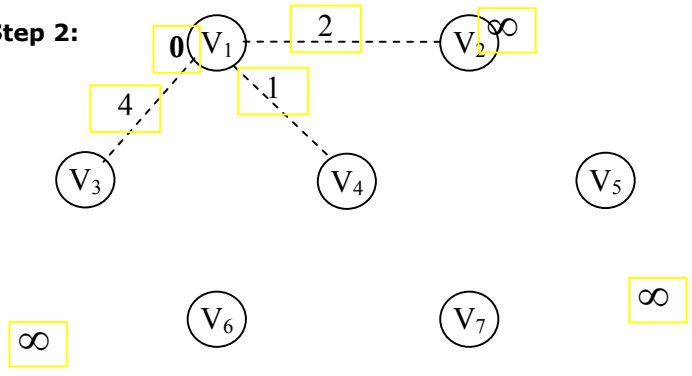


Step 1: Consider V1 as source vertex and proceed further.



Vertex	Initial State		
	known	d_v	P_v
V1	0	0	0
V2	0	∞	0
V3	0	∞	0
V4	0	∞	0
V5	0	∞	0
V6	0	∞	0
V7	0	∞	0

Step 2:



Vertex	After V1 is declared known		
	known	d_v	P_v
V1	1	0	0
V2	0	2	V1
V3	0	4	V1
V4	0	1	V1
V5	0	∞	0
V6	0	∞	0
V7	0	∞	0

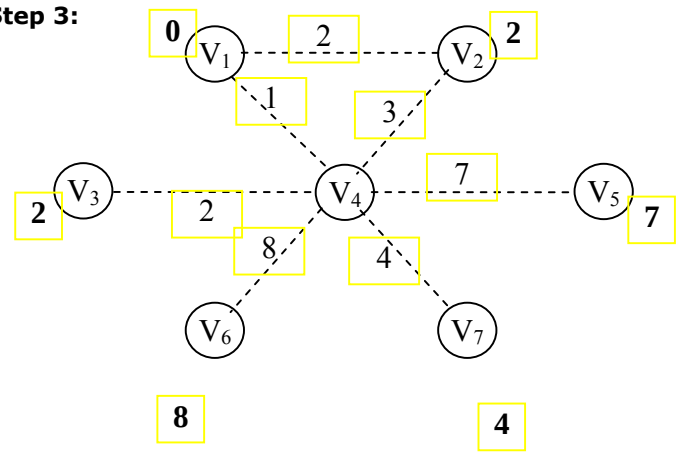
Here 'Vi' is marked as visited and then the distance of its adjacent vertex is updated as follows.

$T[V_2].dist = Min\ [T[V_2].dist, C_{v1, v2}]$ = Min [, 2] = 2.

$T[V_4].dist = Min\ [T[V_4].dist, C_{v1, v4}]$ = Min [∞ , 1] = 1.

$T[V_3].dist = Min [T[V_3].dist, C_{v1, v3}] = Min[\infty, 1] = 4$

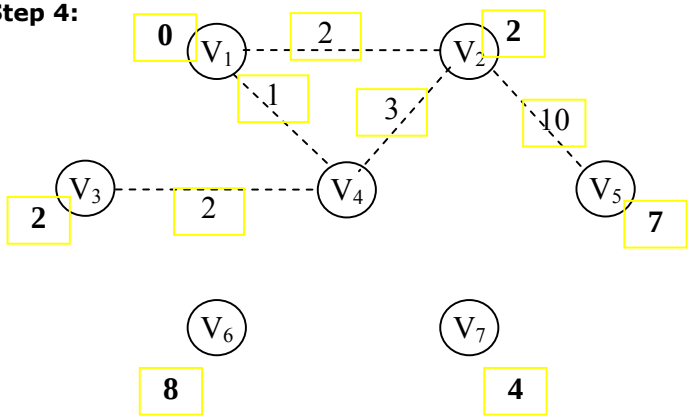
Step 3:



Vertex	After V4 is declared known		
	known	d_v	P_v
V1	1	0	0
V2	0	2	V1
V3	0	2	V4
V4	1	1	V1
V5	0	7	V4
V6	0	8	V4
V7	0	4	V4

Here 'V4' is marked as visited and then the distance of its adjacent vertex is updated.

Step 4:



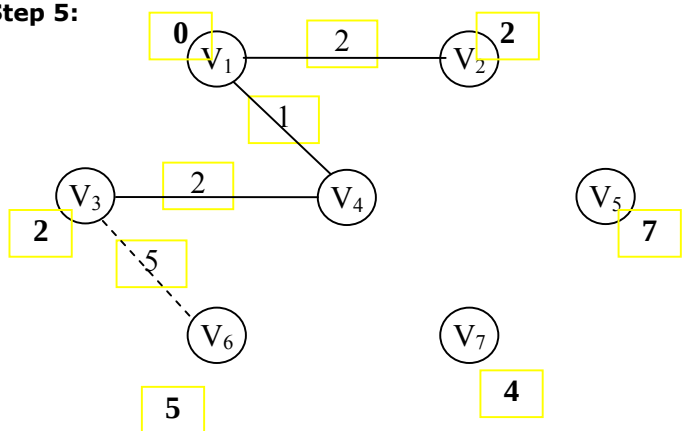
Vertex	After V2 is declared known		
	known	d_v	P_v
V1	1	0	0
V2	1	2	V1
V3	0	2	V4
V4	1	1	V1
V5	0	7	V4
V6	0	8	V4
V7	0	4	V4

Here 'V2' is marked as visited and then the distance of its adjacent vertex is updated as follows;

$T[V_4].dist = Min [T[V_4].dist, C_{v2, v4}] = Min [1, 3] = 1.$

$T[V_5].dist = Min [T[V_5].dist, C_{v2, v5}] = Min [7, 10] = 7$

Step 5:

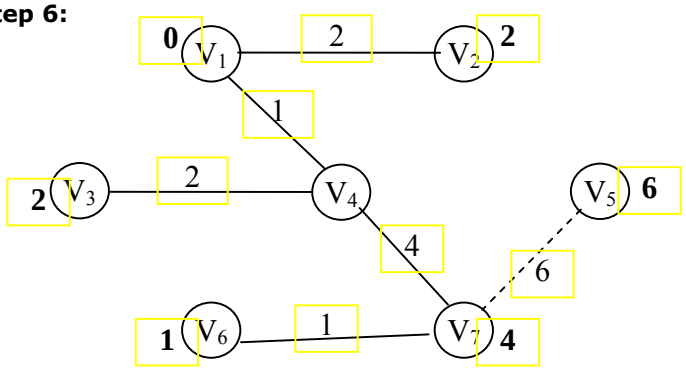


Vertex	After V3 is declared known		
	known	d_v	P_v
V1	1	0	0
V2	1	2	V1
V3	1	2	V4
V4	1	1	V1
V5	0	7	V4
V6	0	5	V3
V7	0	4	V4

Here 'V3' is marked as visited and then the distance of its adjacent vertex is updated as follows;

$T[V_6].dist = Min [T[V_6].dist, C_{v3, v6}] = Min [8, 5] = 5$

Step 6:



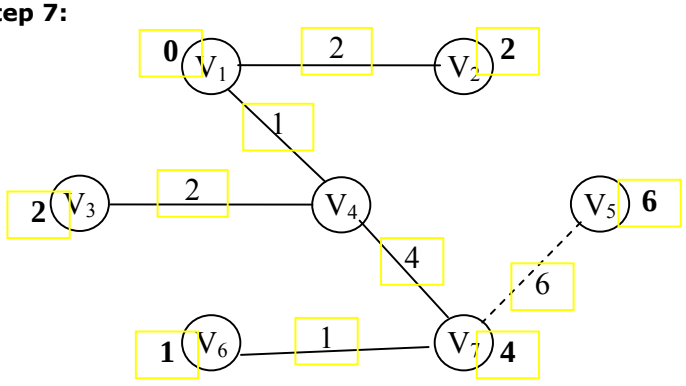
Vertex	After V7 is declared known		
	known	d_v	P_v
V1	1	0	0
V2	1	2	V1
V3	1	2	V4
V4	1	1	V1
V5	0	6	V7
V6	0	1	V7
V7	1	4	V4

Here 'V7' is marked as visited and then the distance of its adjacent vertex is updated as follows;

$T[V_6].dist = Min [T[V_6].dist, C_{v7,v6}] = Min [5, 1] = 1$

$T[V_6].dist = Min [T[V_6].dist, C_{v7,v5}] = Min (7, 6) = 6$

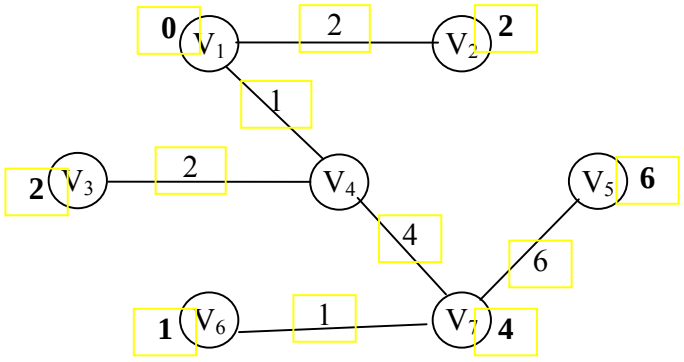
Step 7:



Vertex	After V6 is declared known		
	known	d_v	P_v
V1	1	0	0
V2	1	2	V1
V3	1	2	V4
V4	1	1	V1
V5	0	6	V7
V6	1	1	V7
V7	1	4	V4

Here 'V6' is marked as visited and then the distance of its adjacent vertex is updated.

Step 8:

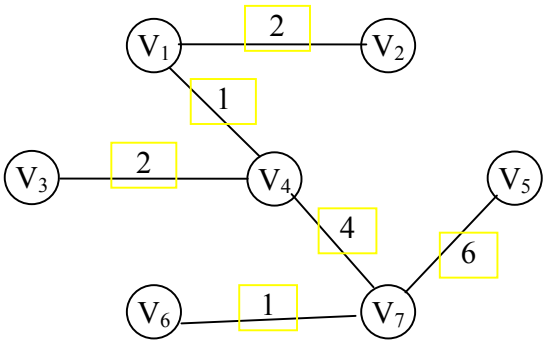


Vertex	After V5 is declared known		
	known	d_v	P_v
V1	1	0	0
V2	1	2	V1
V3	1	2	V4
V4	1	1	V1
V5	1	6	V7
V6	1	1	V7
V7	1	4	V4

Here 'V5' is marked as visited and then the distance of its adjacent vertex is updated.

The final MST is;

Minimum Spanning Tree Cost is **16** for the given Undirected graph.



Implementation of Prim's Algorithm:

```
#include<stdio.h>
#include<conio.h>
void main()
{
        float lcost[8],a[8][8];
        int clost[8],i,j,k,min,n;
        clrscr();
        printf("enter the no. of vertices");
        scanf("%d",&n);
        printf("input weighted matrix");
        for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
        scanf("%f",&a[i][j]);
        for(i=2;i<=n;i++)
        {
                lcost[i]=a[1][i];
                clost[i]=1;
        }
        printf("minimum cost spanning tree edges are:\n");
        for(i=2;i<=n;++i)
        {
                min=lcost[2];
                k=2;
                for(j=3;j<=n;++j)
                if(lcost[j]<min)
                {
                        min=lcost[j];
```

```
                        k=j;
                }
                printf("%d\t%d\n",k,clost[k]);
                lcost[k]=200;
                for(j=2;j<=n;++j)
                        if((a[k][j]<lcost[j])&&(lcost[j]<200))
                        {
                                lcost[j]=a[k][j];
                                clost[j]=k;
                        }
        }
        getch();
}
```

4.7.2 Kruskal's Algorithm:

In Kruskal's algorithm, we select edges inorder of smallest weight and accept an edge if it does not form a cycle. On adding edges to the MST, it should not form cycles.

Initially, there are |V| single node trees. Adding an edge merges two trees into one. When enough edges are accepted, the algorithm terminates with only one tree and this is the minimum spanning tree.

The simple logic is to decide whether edge (u,v) should be accepted or rejected is carried out with Union/Find operation of sets.

The Algorithm(Informal):

i) The edges are build into a minheap structure and each vertex is considered as a single node tree.

ii) The DeleteMin operation is utilized to find the minimum cost edge(u,v).

iii) The vertices u and v are searched in the spanning tree set S and if the returned sets are not same then (u,v) is added to the set S (union operation is performed), with the constraint that adding (u,v) will not create a cycle in spanning tree set S.

iv) Repeat step (ii) & (iii), until a spanning tree is constructed with |v|-1 edges.

Algorithm:

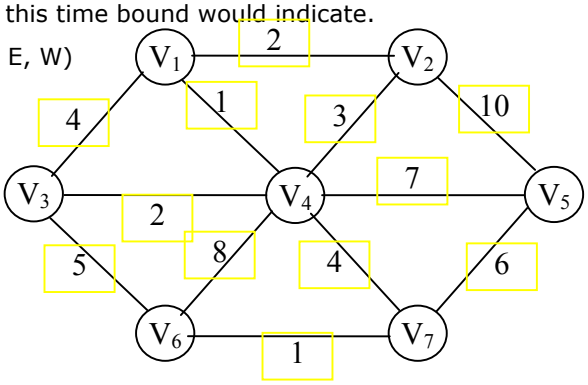
```
void Kruskal(Graph G)
{
 int EdgesAccepted;
 Disjset S;
 PriorityQueue H;
 Vertex U,V;
 SetType Uset, Vset;
 Edge E;
 Initialize(S);
 ReadGraphIntoHeapArray(G,H);
 BuildHeap(H);
```

```
        EdgesAccepted = 0;
        While(  EdgesAccepted < NumVertex-1)
        {
                E = DeleteMin(H);               /* E = (U,V)*/
                Uset = Find(U,S);
                Vset = Find(V,S);
                if(Uset!=Vset)
                {
                        /* Accep the Edge*/
                        EdgesAccepted++;
                        SetUnion(S,Uset,Vset);
                }
        }
        }
```

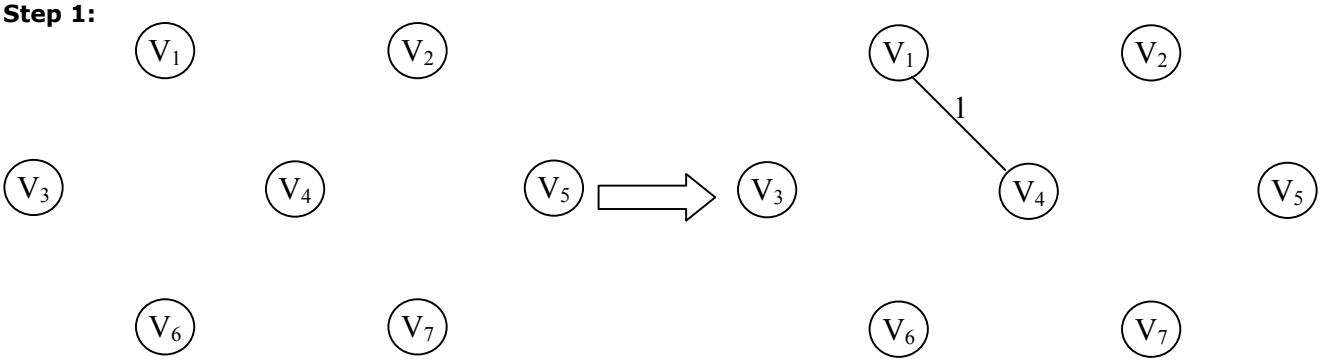
Analysis of the Kruskal's Algorithm:

The worst case running time of this algorithm is O(|E| log |E|), which is dominated by heap operations. Notice that since $|E| = O(|V|^2)$, this running time is actually O(|E| log |V|). In practice, the algorithm is mush faster than this time bound would indicate.

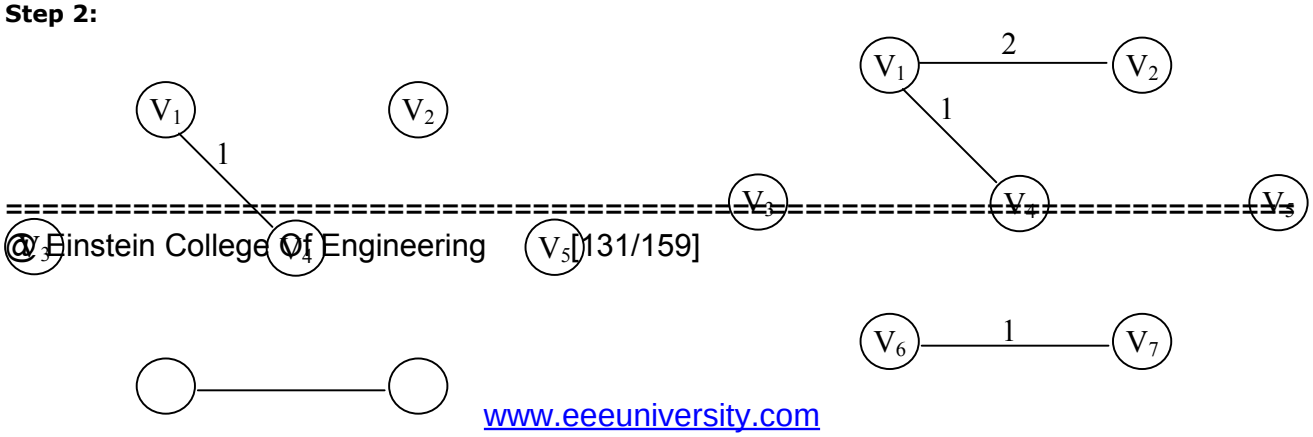
Example: Given G = (V, E, W)

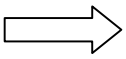


Step 1:

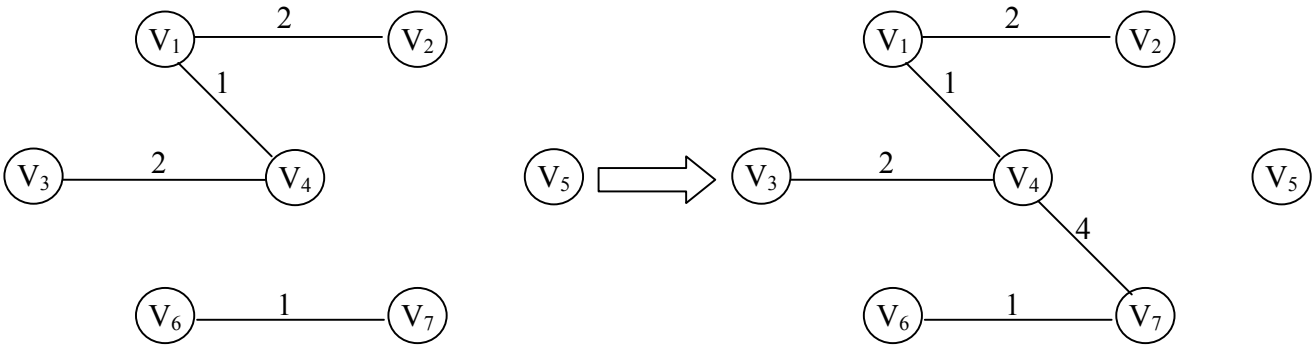


Step 2:

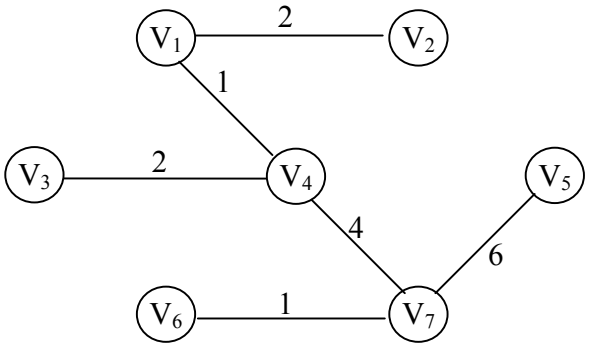




Step 3:



Step 4: The final MST using kruskal's algorithm is as follows:



The final Execution table for finding MST using Kruskal's algorithm is as follows:

Edge	Weight	Action
(V1, V4)	1	Accepted
(V6, V7)	1	Accepted
(V1, V2)	2	Accepted
(V3, V4)	2	Accepted
(V2, V4)	3	Rejected
(V1, V3)	4	Rejected
(V4, V7)	4	Accepted
(V3, V6)	5	Rejected
(V5, V7)	6	Accepted

4.8 BICONNECTIVITY

A connected undirected graph G is biconnected if there are no vertices whose removal disconnects the rest of the graph.

A connected undirected graph G is said to be biconnected if it remains connected after the removal of any one vertex and the edges that are incident upon that vertex.

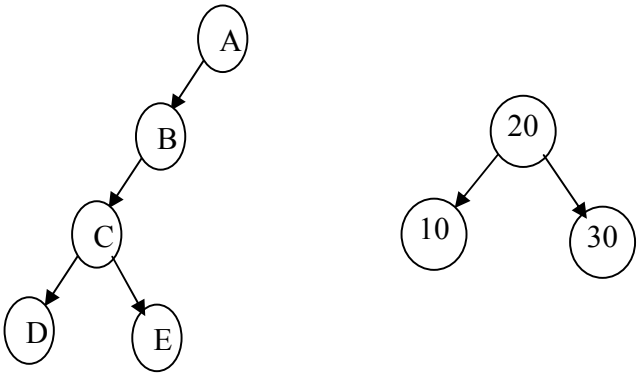
4.8.1 Biconnected Component:

A biconnected component (bicomponent) of an undirected graph is a maximal biconnected subgraph, i.e. a biconnected subgraph not contained in any larger biconnected subgraph.

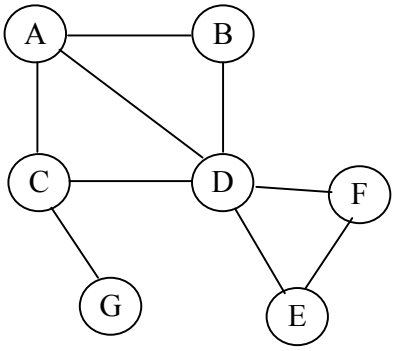
4.8.2 Articulation Points

The vertices whose removal would disconnect the graph are known as articulation points.

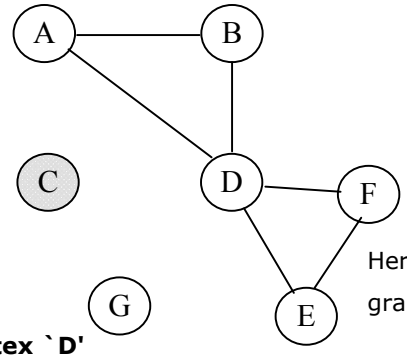
Example: Let us consider a Connected Undirected Graph



4.8.3 Explanation with an Example: Figure 4.8.1

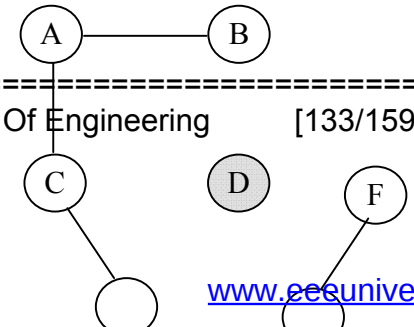


Removal of vertex `C'



Here the removal of `C' vertex will disconnect G from the graph.

Removal of vertex `D'



IIIly removal of `D' vertex will disconnect E & F from the graph. Therefore `C' & `D' are articulation points.

The graph is not biconnected, if it has articulation points.

Depth first search provides a linear time algorithm to find all articulation points in a connected graph.

Steps to find Articulation Points :

Step 1 : Perform Depth first search, starting at any vertex

Step 2 : Number the vertex as they are visited, as Num (v).

Step 3 : Compute the lowest numbered vertex for every vertex v in the Depth first spanning tree, which we call as low (w), that is reachable from v by taking zero or more tree edges and then possibly one back edge. By definition, Low(v) is the minimum of

(i) Num (v)

(ii) The lowest Num (w) among all back edges (v, w)

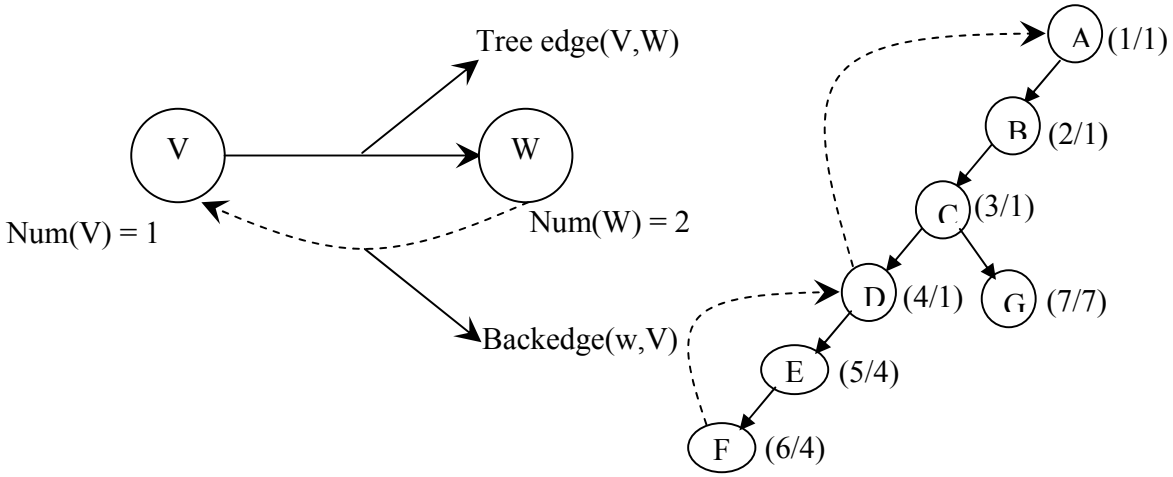
(iii) The lowest Low (w) among all tree edges (v, w)

Step 4 : (i) The root is an articulation if and only if it has more than two children.

(ii) Any vertex v other than root is an articulation point if and only if v has same child w such that Low (w) Num (v), The time taken to compute this algorithm an a graph is

Note:

For any edge (v, w) we can tell whether it is a tree edge or back edge merely by checking Num (v) and Num (w). If Num (w) > Num (v) then the edge (v, w) is a back edge.



Depth First Tree For Fig (4.8.1) With Num and Low.

ROUTINE TO COMPUTE LOW AND TEST FOR ARTICULATION POINTS

```
void AssignLow (Vertex V)
{
    Vertex W;
    Low [V] = Num [V]; /* Rule 1 */
    for each W adjacent to V
    {
        If (Num [W] > Num [V]) /* forward edge */
        {
            Assign Low (W);
            If (Low [W]> = Num [V])
            Printf ("% V is an articulation pt \n", V);
            Low[V] = Min (Low [V], Low[V]); /* Rule 3*/
        }
        Else
        If (parent [V] ! = W) /* Back edge */
        Low [V] = Min (Low [V], Num [W])); /* Rule 2*/
    }
}
```

Low can be computed by performing a postorder traversal of the depth - first spanning tree. (ie)

Low (F) = Min (Num (F), Num (D))

/* Since there is no tree edge & only one back edge */

= Min (6, 4) = 4

Low (F) = 4

Low (E) = Min (Num (E), Low (F))

/* there is no back edge */.

= Min (5, 4) = 4

Low (D) = Min (Num (D), Low (E), Num (A))

= Min (4,4,1) = 1

Low (D) = 1

Low (G) = Min (Num (G)) = 7 /* Since there is no tree edge & back edge */

Low (C) = Min (Num (C), Low (D), Low (G))

= Min (3,1,7) = 1

Low (C) = 1 .

IIIly Low (B) = Min (Num (B), Low (C))

= Min (2,1) = 1

Low (A) = Min (Num (A), Low (B))

= Min (1, 1) = 1

Low (A) = 1.

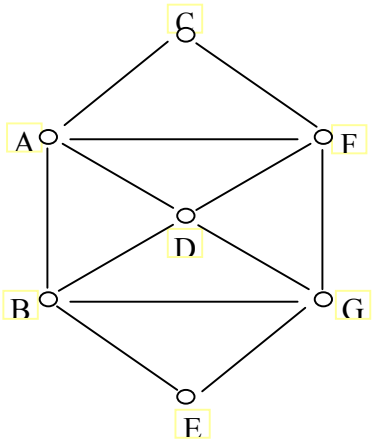
From fig (4.8) it is clear that Low (G) > Num (C) (ie) 7 > 3 /* if Low (W) Num (V)*/ the `v' is an articulation pt Therefore `C' is an articulation point.

III[ly] Low (E) = Num (D), Hence D is an articulation point.

4.8.4 Euler Circuit:

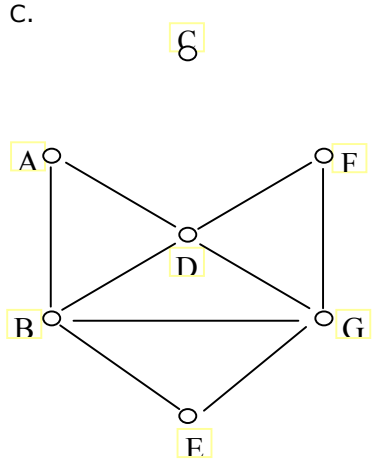
- ❖ Visiting all the edges only one with different start and end vertex is a path, otherwise it is a circuit.
- ❖ If the condition "Each vertex has an even degree and the graph is connected", is satisfied, then the graph have an Euler circuit.
- ❖ DFS considers each edge once and all vertices while Euler visit them.
- ❖ DFS visits a vertex only once, but Euler circuit may visit more than once.
- ❖ Suppose we make DFS visit edges in order that it considers, derives a Euler circuit.

Example:

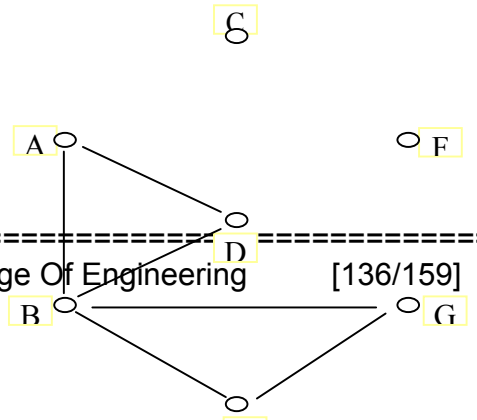


- ❖ Suppose we start from C, we would go C, A, F, C

Now we stuck in C.



- ❖ Now backtrack to F and process F, D, G, F. We now stuck in F.



Splice the paths CAFC and FDGF, we get C, A, F, D, F, C.

- Now backtrack to G and process G, E, B, A, D, B, G.

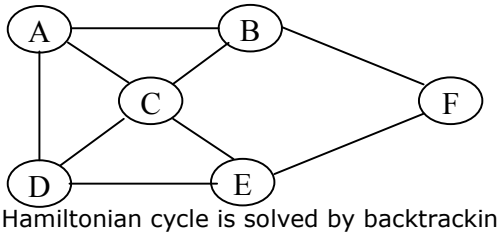
  Splice the path with previous, we get:

  C, A, F, D, G, E, B, A, D, B, G, F, C

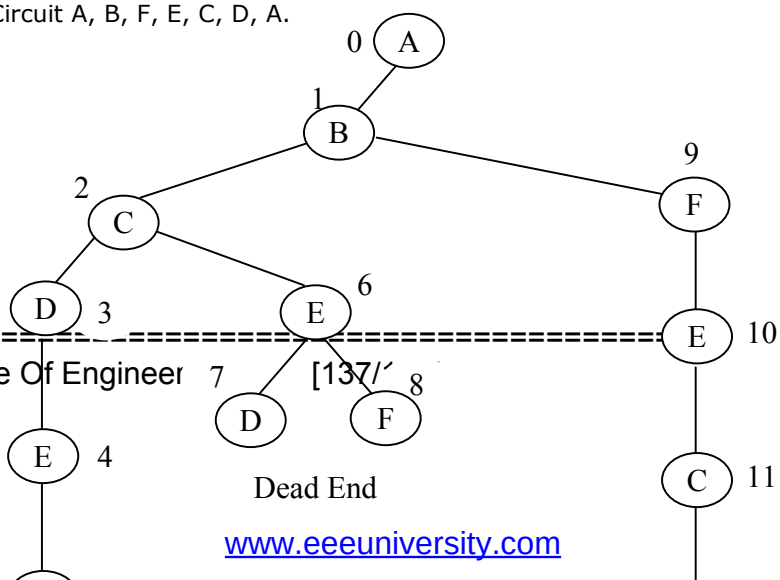
As a result no more edges are left unvisited and we have obtained an Euler circuit, having a running time O(e+n).

4.8.5 Hamiltonian Circuit Problem:

- A Hamiltonian path is a path in an undirected graph which visits each vertex exactly once and also returns to the starting vertex.
- Determining whether such paths and cycles exist in graphs is the Hamiltonian path problem which is NP-complete.
- These paths and cycles are named after William Rowan Hamiltonian.
- Consider a Graph G;



- The problem of finding Hamiltonian cycle is solved by backtracking approach.
- We consider the vertex 'A' as the root. From vertex 'A' we have three ways, so we resolve the tie using alphabet order, so we select vertex 'B'.
- From 'B' algorithm proceeds to 'C', then to 'D', then to 'E' and finally to 'F' which proves to be a dead end.
- So the algorithm backtracks from 'F' to 'E', then to 'D' and then to 'C', which provides alternative to pursue.
- Going from 'C' to 'E' which is useless, and the algorithm has to backtrack from 'E' to 'C' and then to 'B'. From there it goes to the vertices 'F', 'E', 'C' & 'D', from which it return to 'A', yielding the Hamiltonian Circuit A, B, F, E, C, D, A.



UNIT V :ALGORITHM DESIGN AND ANALYSIS

Greedy algorithms – Divide and conquer – Dynamic programming – backtracking – branch and bound – Randomized algorithms – algorithm analysis – asymptotic notations – recurrences – NP-complete problems

Unit V

5.1 Greedy Algorithm

Greedy algorithms work in phases. In each phase, a decision is made that appears to be good, without regard for future consequences(cost). This means that some *local optimum* is chosen. This "take what you can get now" strategy is the source of the name for this class of algorithms. When the algorithm terminates, we hope that the local optimum is equal to the *global optimum*. If this is the case, then the algorithm is correct; otherwise, the algorithm has produced a *suboptimal solution*. If the absolute best answer is not required, then simple greedy algorithms are sometimes used to generate approximate answers, rather than using the more complicated algorithms generally required to generate an exact answer.

To make a greedy algorithm, we have to concentrate on two sets:

1. One for candidate chosen
2. Another for candidates that are not chosen

Add the candidate we chose to the *chosen set* if we have a solution, else we iterate through loop and get the next optimal candidate.

- Greedy algorithms are simple and straightforward.
- They are shortsighted in their approach in the sense that they take decisions on the basis of information at hand without worrying about the effect these decisions may have in the future.
- They are easy to implement, invent and efficient. They are used to solve optimization problems.

5.1.1 Greedy approach for Making Change Problem:

Greedy algorithm works by making the decision that seems most promising at any moment. It never reconsiders this decision, whatever situation may arise later. Consider the problem of "Making Change":

Coins available are:

- Dollars (100 cents)
- Quarters (25 cents)

- Dimes (10 cents)
- Nickels (5 cents)
- Pennies (1 cent)

The problem is to make a change of a given amount using the smallest possible number of coins.

The informal algorithm is:

- Start with nothing
- At every stage without passing the given amount.
- Add the largest to the coins already chosen.

Formal algorithm for make change for n units using the least possible number of coins:

```
Make-Change (n)
        C ← {100, 25, 10 , 5, 1};              // constant
        Sol← {};                                // Set that will hold the solution set
        Sum ← 0                                 // sum of item in solution set
        WHILE sum not = n
                x = largest item in set C such that sum + x = n
                IF no such item THEN
                        RETURN "No Solution"
                S← S{value of x}
                Sum← sum + x
        RETURN S
End;
```

The problem asks to provide a change for a specified amount, using a minimal number of coins. A greedy algorithm may at each stage employ the criteria of providing the largest available coin which is not greater than the amount still owed

To construct the solution in an optimal way, let us see how the *greedy algorithm* works:

- Algorithm contains two sets; 1. Chosen items 2. Rejected items
- Consists of four functions:
  - A function that checks whether chosen set of items provides a solution.
  - A function that checks the feasibility of a set.
  - The selection function tells which of the candidate is the most promising.
  - An objective function, which does not appear explicitly, gives the value of a solution.

The greedy algorithm is shown below:

```
procedure GreedyMaking Change
    Initially the set of chosen items is empty
        i.e solution set
    At each step
        item will be added in a solution set by using selection function.
    IF the set would be longer be feasible
        reject them under consideration and is never consider again.
    ELSE IF set is still feasible THEN
```

add the current ITEM.

End; // Procedure GreedyMaking Change

5.1.2 Egyptian Fractions:

An **Egyptian fraction** is the sum of *distinct unit fractions*, such as $\frac{1}{2}+\frac{1}{3}+\frac{1}{16}$. That is, each fraction in the expression has a numerator equal to 1 and a denominator that is a positive integer, and all the denominators differ from each other.

The sum of an expression of this type is a *positive rational number* a/b; for instance the Egyptian fraction above sums to 43/48. Every positive rational number can be represented by an Egyptian fraction.

A greedy algorithm, for finding such a representation, can add at each stage to the sum of the largest unit fraction which does not cause the sum to exceed the fraction.

Ex: $$\frac{97}{110}=\frac{1}{2}+\frac{1}{5}+\frac{1}{11}$$

5.1.3 Map Coloring: The map coloring problem asks to assign colors to the different regions, where adjacent regions are not allowed to have the same color. There is no general algorithm to assign minimal number of colors, yet each map has an assignment which uses no more than four colors.

The greedy approach, repeatedly choose a new color and assign it to as many regions as possible.

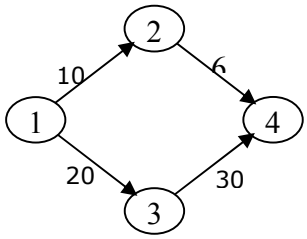
5.1.4 Shortest Path Algorithm:

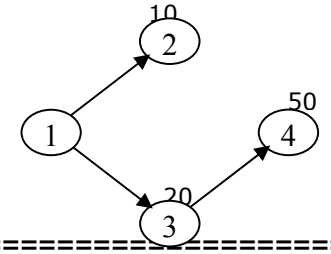
Establish the shortest path between a single source node and all of the other nodes in a graph.

Greedy algorithm: Add an edge that connects a new node to those already selected. The path from the source to the new node should be the shortest among all the paths to nodes that have not been selected yet.

Greedy Approach:

- We need an array to *hold the shortest distance* to each node, and a *visited set*.
- We take neighboring node; get the direct distance from it to the root node.
- Next we have to check *if it is less than the entry in the array* or *if it not less than the entry in the array* or *if the array has null value,* then we store it in the array.
- From that node, put the neighboring node in the visited set. Then we visit every other node it is connected to and also calculate the distance from it to the root node.
- *If it is less than the entry in the array* or *if the array has null value,* then we store it in the array.
- After we are finished with that node we go to the next connected node, and so on.
- At the end we will have an array of values representing the shortest distance from the starting node to every other node in the graph.
- This algorithm is also known as Dijkstra's algorithm.





```
Algorithm ShortestPaths(v, cost, dist, n)
// dist[j], 1≤j ≤ n, is set to the length of the shortest path from vertex v to vertex j in a digraph G
with n vertices. Dist[v] is set to zero. G is represented by its cost adjacency matrix cost[1:n, 1:n] //
{
 for i = 1 to n do
 {
        S[i] = false;                    // Initialize S
        dist[i] = cost[v,i];
 }
 S[v] = true;
 dist[v] = 0.0;                          // Put v in S
 for num = 2 to n do
 {
// Determine n-1 paths from u. Choose u from among those vertices not in S such that dist[u] is minimum;
        S[u] = true;                                                    //  Put
    u in S
        for (each w adjacent to u with S[w] = false) do
        if (dist[w] > (dist[u] + cost[u,w])) then          // Update distances
         dist[w] = dist[u] + cost[u,w];
    }
}
```

5.1.5 Greedy approach for Knapsack Problem:

Description: Given a set of items each with a cost and a value, determine the number of each item to include in a collection so that the total cost does not exceed some given cost and the total value is as large as possible.

We are given n objects and a knapsack. Object i has a weight w_i and the knapsack has a capacity m. The problem is that a thief robbing a store and can carry a maximal weight of w into their knapsack. What items should thief take?

There are two version of problem:

1. Fractional knapsack problem by Greedy Algorithm which solves the problem by putting items into the knapsack one-by-one. This approach is greedy because once an item has been put into the knapsack, it is never removed.
2. 0 – 1 knapsack problem by Dynamic programming algorithm

We can apply greedy approach to solve it.

The Greedy solution to the Fractional Knapsack problem is;

1. If a fraction x_i, $0 \le x_i \le 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned.
2. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most m.
3. Formally, the problem can be stated as

maximize $\sum_{1\le i\le n} p_i x_i$

subject to $\sum_{1\le i\le n} w_i x_i \le m$

and $0 \le x_i \le 1, 1 \le i \le n$

4. The profits and weights are positive numbers.

Algorithm:

```
Algorithm GreedyKnapsack(m,n)
// p[1:n] and w[1:n] contain the profits and weights respectively of the n objects ordered
//such that p[i]/w[i] ≥ p[i+1]/w[i+1]. m is the knapsack size and x[1:n] is the solution vector.
{
    for i = 1 to n do x[i] = 0.0;                    // Initialize x
    U = m;
    for i =1 to n do
    {
        if (w[i] > U) then break;
        x[i] = 1.0;
        U = U – w[i];
    }
    if(i≤n) then x[i] = U/w[i];
}
```

Procedure

```
Procedure GreedyFractionalKnapsack(w, v, W)
FOR i = 1 yo n do
    x[i] = 0
weight=0
while(weight<W)
    do i = best remaining item
        If weight + w[i] = W
            then x[i] = 1
                weight = weight + w[i]
            else
                x[i] = (w- weight) / w[i]
                weight = W
    return x
End;
```

- If the items are already sorted in the descending order of v_i/w_i, then the while-loop takes a time in O(n).

- Therefore, the total time including the sort is O(n log n)
- If we keep the items in heap with largest v_i/wi at the root. Then creating the heap takes O(n) time.
- While loop now takes O(log n) time.

The optimal solution to this problem is to sort by the value of the item in decreasing order. Then pick up the most valuable item which also has a least weight.

First, if its weight is less than the total weight that can be carried, then deduct the total weight that can be carried by the weight of the item just pick.

The second item to pick is the most valuable item among those remaining.

5.2 Divide and Conquer

- It is a Top-Down technique for designing algorithms.
- Divide and Conquer algorithms consists of two parts:

  1. **Divide**: Dividing the problem into Smaller sub-problems, which are solved recursively.
  2. **Conquer**: The solution to the original problem is then formed from the solutions to the sub-problems

The divide and conquer paradigm consists of following major phases:

1. Breaking the problem into several sub-programs that are similar to the original problem but smaller in size.
2. Solve the sub-problem recursively (successively and independently)
3. Combine these solutions of sub-problems to create a solution to the original problem.

5.2.1 Binary Search by divide and conquer algorithm

- For an given array of n elements, the basic idea of binary search is that for a given element we "probe" the middle element of the array.
- We continue in either lower or upper segment of the array, depending on the outcome of the probe until we reached the required element.

Algorithm

1. Let A[1…n] be an array of non-decreasing sorted order.
2. Let 'q' be the query point.
3. The problem consists of finding 'q' in the array. If 'q' is not in A, then find the position where 'q' might be inserted.
4. Formally, find the index i such that $1 \le i \le n+1$ and $A[i-1] < x < A[i]$.
5. Look for 'q' either in the first half or in the second half of the array A.
6. Compare 'q' to an element in the middle, n/2 of the array. Let k = n/2.
7. If q < A[k], then search in the A[1..k].
8. Otherwise search A[k+1……n] for 'q'.

Recursive call to the function and the procedure

/* Recursive procedure for Binary search*/

BinarySearch(A[0…n-1], value, low, high)

{

```
        if(high < low)
                return – 1              // not found
        mid = (low + high)/2
        if (A[mid]>value)
                return BinarySearch(A, value, low, mid-1)
        else if(A[mid] < value)
                return binarySearch(A, value, mid+1, high)
        else
                return mid              // found
       }
```

We can eliminate the recursion by initializing the low = 0 and high = n-1

```
       /* Non-recursive procedure for Binary search */
       BinarySearch(A[0….n-1], value)
       {
       low = 0, high = n-1;
       while (low <= high)
       {
                mid = (low + high) /2
                if (A[mid] > value)
                        high = mid -1
                else if (A[mid < value)
                        low = mid +1
                else
                        return mid              // found
       }
       return -1                               // not found
       }
```

5.2.2 Merge Sort - Divide and conquer algorithm

- DIVIDE: Partition the n- element sequence to be sorted into two subsequences of n/2 elements each.
- CONQUER: Sort the two subsequences recursively using the mergesort.
- COMBINE: Merge the two sorted subsequences of size n/2 each to produce the sorted sequence consisting of n elements.

Overall process

1. Split the original list into two halves.
2. Sort each half using merge sort
3. Merge the two sorted halves together into a single sorted list.

```
procedure mergesort(first, last, array)
        mid = (first + last)/2
```

mergsort(first, mid, array)

mergsort(mid+1, last, array)

rejoin_two_halves(mid, array)

end mergesort

The time complexity of the algorithm satisfies the recurrence equation

- T(n) = 2T(n/2) + O(n) → if n>1
- T (n) = 0 → if n=1

whose solution is T(n) = O (n log n).

5.2.3 Quick Sort – Divide and conquer algorithm

Here recombine step requires no additional work.

1. The essence of quick sort is to sort an array A[i] ….A[n] by picking some key value 'v' in the array as pivot element around which to rearrange the elements in the array.
2. The pivot is near the median key value in the array, so that it is preceded by about half the keys and followed by about half.
3. We process the execution of elements in the array so that for some j, all the records with keys less than 'v' appears in A[1], A[2] ….. A[j] and all those with keys 'v' or greater than 'v' appear in A[j+1] ..A[n]
4. Apply quick sort to the newly formed array A[1]… A[j] and A[j+1] …… A[n].
5. Since all the keys in the first group precede all the keys in the second group, thus the entire array be sorted.
6. Sets of cardinality greater than one are partitioned into two subsets, and then the algorithm is recursively invoked on the subsets.
7. The partitioning uses a key from the set as pivot.
8. Values smaller than the pivot are sent one subset, and values greater than the pivot are sent to the other subset.
9. For randomly chosen pivots, the expected running of the algorithm satisfies recurrence equation.

$$T(N) = T(i) + T(N-j-i) + CN$$

- The running time of quick sort is equal to the running time of the two recursive calls plus the linear time spent in the partition.
- The pivot selection takes constant time only.

Algorithm:

```
void quicksort(int s[], int l, int h)
{
   int p; /* index of partition */
   if ((h - l) > 0) {
      p = partition(s, l, h);
      quicksort(s, l, p - 1);
      quicksort(s, p + 1, h);
   }
```

```
}
int partition(int s[], int l, int h)
{
   int i;
   int p; /* pivot element index */
   int firsthigh; /* divider position for pivot element */
   p = h;
   firsthigh = l;
   for (i = l; i < h; i++)
      if(s[i] < s[p]) {
         swap(&s[i], &s[firsthigh]);
         firsthigh++;
      }
   swap(&s[p], &s[firsthigh]);
   return(firsthigh);
}
void swap(int *a, int *b)
{
      int x;
      x = *a;
      *a = *b;
      *b = x;
}
```

Analysis of Quick sort:

Worst case → T(N) = O(N2)

Best case → T(N) = O(N log N)

Average case → T(N) = O(N log N)

5.2.4 Matrix Multiplication – Divide and Conquer Algorithm

- The given problem has two N by N matrices A and B and we have to compute C = AxB.

$$\begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} X \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$$

- The divide and conquer method for performing the multiplication is as shown below:

P1 = (A11 + A22) (B11 + B22)

P2 = (A21+A22)B11

P3 = A11(B12-B22)

P4 = A22(B12-B22)

P5 = (A11+A12)B22

P6 = (A21-A11) (B11+B12)

P7 = (A12-A22) (B21 + B22)

C11 = P1 + P4 – P5 + P7

C12 = P3 + P5

C21 = P2 + P4

C22 = P1 + P3 – P2 + P6

T(n) = 7T(n/2) + cn → if n>1

T(n) = c → if n =1

T(n) = O(nlog 7) = O(n2.81)

5.4 Dynamic programming

- Dynamic programming is a method of solving problems exhibiting the properties of overlapping subproblems and optimal substructure which takes less time.
- By this way a solutions may be viewed as a result of a sequence of decisions.
- Dynamic programming takes advantage for the duplication and arranges to solve each subproblem only once, saving the solution for later use.
- The idea of dynamic programming is to avoid calculating the same stuff twice.
- A dynamic programming is a bottom-up approach. It means;
  1. Start with the smallest sub problems.
  2. Combining their solutions, obtain the solutions to subproblems of increasing size.
  3. Until arrive at the solution of the original problem.

Basic idea of dynamic programming

- Start with an inefficient recursive algorithm.
- Speed it up by only solving each subproblem once and remembering the solution later use.
- May rearrange the order of subproblem computation to be more convenient.

Principle of Optimality

- The dynamic programming relies on a principle of optimality, which states that ***an optimal sequence of decisions or choices has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision***.
- Thus, the essential difference between the greedy method and dynamic programming is that in the greedy method only one decision sequence is ever generated. In dynamic programming, many decision sequences may be generated. However, sequences containing suboptimal subsequences cannot be optimal(it the principle of optimality holds) and so will not be generated.
- Ex, in matrix chain multiplication problem
- The principle can be related as follows: the optimal solution to a problem is a combination of optimal solutions to some of its subproblems.
- The difficulty in turning the principle of optimality into an algorithm is that it is not usually obvious which subproblems are relevant to the problem under consideration.

5.4.1 0-1 knapsack problem

- The problem statement is that a thief robbing a store and can carry a maximal weight of W into their knapsack.
- There are n items and i^{th} item weight w_i and is worth v_i dollars.
- What items should thief take? There are two versions of problem;

1. Fractional knapsack problem which dealed in greedy algorithm.
2. 0 – 1 knapsack problem:

The setup is same, but the items may not be broken into smaller pieces, so the thief decide either to take an item or to leave it(binary choice), but may not take a fraction of an item. 0-1 knapsack problem has the following properties:

1. Exhibit no greedy choice property.
2. Exhibit optimal substructure property.
3. Only dynamic programming algorithm exists.

Step1: Structure: Characterize the structure of an optimal solution.

– Decompose the problem into smaller problems, and find a relation between the structure of the optimal solution of the original problem and the solutions of the smaller problems.

Step2: Principle of Optimality: Recursively define the value of an optimal solution.

– Express the solution of the original problem in terms of optimal solutions for smaller problems.

Step 3: Bottom-up computation: Compute the value of an optimal solution in a bottom-up fashion by using a table structure.

Step 4: Construction of optimal solution: Construct an optimal solution from computed information.

Steps 3 and 4 may often be combined.

Remarks on the Dynamic Programming Approach

- Steps 1-3 form the basis of a dynamic-programming solution to a problem.
- Step 4 can be omitted if only the value of an optimal solution is required.

Developing a DP Algorithm for Knapsack

Step 1: Decompose the problem into smaller problems.

We construct an array $V[0..n, 0..W]$ For $1 \le i \le n$, and $0 \le w \le W$, the entry V[i,W] will store the maximum (combined) computing time of any subset of files (1,2..i)ined) size at most w.If we can compute all the entries of this array, then the array entry V[n.W] will contain the maximum computing time of files that can fit into the storage, that is, the solution to our problem.

Step 2: Recursively define the value of an optimal solution in terms of solutions to smaller problems.

Initial Settings: Set

$$V[0,w] = 0 \quad \text{for } 0 \le w \le W, \quad \text{no item}$$
$$V[i,w] = -\infty \quad \text{for } w < 0, \quad \text{illegal}$$

Recursive Step: Use

$$V[i,w] = \max(V[i-1,w], v_i + V[i-1, w-w_i])$$

for $1 \le i \le n, 0 \le w \le W$.

Correctness of the Method for Computing $V[i, w]$

Lemma: For $1 \leq i \leq n$, $0 \leq w \leq W$,

$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w-w_i])$.

Proof: To compute $V[i, w]$ we note that we have only two choices for file i:

Leave file i: The best we can do with files $\{1, 2, \ldots, i-1\}$ and storage limit w is $V[i-1, w]$.

Take file i (only possible if $w_i \leq w$): Then we gain v_i of computing time, but have spent w_i bytes of our storage. The best we can do with remaining files $\{1, 2, \ldots, i-1\}$ and storage $(w - w_i)$ is $V[i-1, w-w_i]$.
Totally, we get $v_i + V[i-1, w-w_i]$.

Note that if $w_i > w$, then $v_i + V[i-1, w-w_i] = -\infty$ so the lemma is correct in any case.

The Dynamic Programming Algorithm

```
KnapSack(v, w, n, W)
{
  for (w = 0 to W) V[0, w] = 0;
  for (i = 1 to n)
    for (w = 0 to W)
      if (w[i] ≤ w)
        V[i, w] = max{V[i-1, w], v[i] + V[i-1, w-w[i]]};
      else
        V[i, w] = V[i-1, w];
  return V[n, W];
}
```

Time complexity: Clearly, $O(nW)$.

Constructing the Optimal Solution

The algorithm for computing V[I,W] described in the previous slide does not keep record of which subset of items gives the optimal solution.

To compute the actual subset, we can add an auxiliary boolean array keep[i,w] which is 1 if we decide to take the i^{th} file in V[i,W] and 0 otherwise.

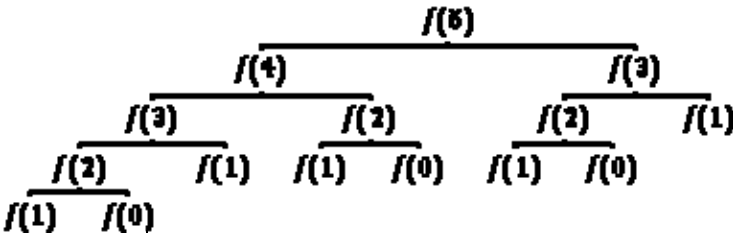
The Complete Algorithm for the Knapsack Problem

```
KnapSack(v, w, n, W)
{
    for (w = 0 to W) V[0, w] = 0;
    for (i = 1 to n)
        for (w = 0 to W)
            if ((w[i] ≤ w) and (v[i] + V[i − 1, w − w[i]] > V[i − 1, w]))
            {
                V[i, w] = v[i] + V[i − 1, w − w[i]];
                keep[i, w] = 1;
            }
            else
            {
                V[i, w] = V[i − 1, w];
                keep[i, w] = 0;
            }
    K = W;
    for (i = n downto 1)
        if (keep[i, K] == 1)
        {
            output i;
            K = K − w[i];
        }
    return V[n, W];
}
```

5.4.2 Fibonnaci Numbers

$$f(i) = \begin{cases} f(i-1) + f(i-2) & \text{if } i > 1 \\ 1 & \text{otherwise} \end{cases}$$

A top-down approach of computing, say, f(5) is inefficient do to repeated subcomputations.



A bottom-up approach computes f(0), f(1), f(2), f(3), f(4), f(5) in the listed order.

Greedy Approach VS Dynamic Programming (DP)

- Greedy and Dynamic Programming are methods for solving optimization problems.
- Greedy algorithms are usually more efficient than DP solutions.
- However, often you need to use dynamic programming since the optimal solution cannot be guaranteed by a greedy algorithm.

- DP provides efficient solutions for some problems for which a brute force approach would be very slow.
- To use Dynamic Programming we need only show that the principle of optimality applies to the problem.

5.5 Backtracking Algorithm:

Backtracking is a general algorithm for finding all (or some) solutions to some computational problem, that incrementally builds candidates to the solutions, and abandons each partial candidate *c* ("backtracks") as soon as it determines that *c* cannot possibly be completed to a valid solution.

The classic textbook example of the use of backtracking is the eight queens puzzle, that asks for all arrangements of eight queens on a standard chessboard so that no queen attacks any other. In the common backtracking approach, the partial candidates are arrangements of *k* queens in the first *k* rows of the board, all in different rows and columns. Any partial solution that contains two mutually attacking queens can be abandoned, since it cannot possibly be completed to a valid solution.

Backtracking can be applied only for problems which admit the concept of a "partial candidate solution" and a relatively quick test of whether it can possibly be completed to a valid solution. It is useless, for example, for locating a given value in an unordered table. When it is applicable, however, backtracking is often much faster than brute force enumeration of all complete candidates, since it can eliminate a large number of candidates with a single test.

a) Solving a maze

- Given a maze, find a path from start to finish
- At each intersection, you have to decide between three or fewer choices:
  - Go straight
  - Go left
  - Go right
- You don’t have enough information to choose correctly
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many types of maze problem can be solved with backtracking

b) Coloring a map

- You wish to color a map with not more than four colors red, yellow, green, blue
- Adjacent countries must be in different colors
- You don’t have enough information to choose colors
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many coloring problems can be solved with backtracking

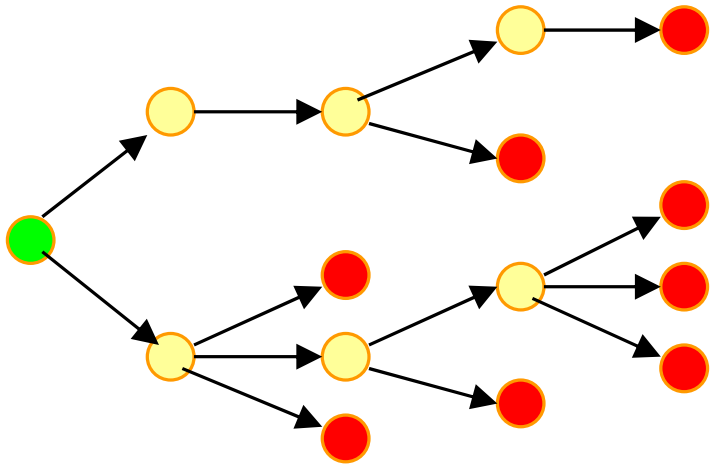
c) Solving a puzzle

- In this puzzle, all holes but one are filled with white pegs
- You can jump over one peg with another
- Jumped pegs are removed

- The object is to remove all but the last peg
- You don't have enough information to jump correctly
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many kinds of puzzle can be solved with backtracking

Terminology I

A tree is composed of nodes



The backtracking algorithm

- Backtracking is really quite simple--we "explore" each node, as follows:
- To "explore" node N:

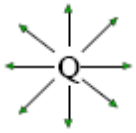
1. If N is a goal node, return "success"
2. If N is a leaf node, return "failure"
3. For each child C of N,
    3.1. Explore C
        3.1.1. If C was successful, return "success"
4. Return "failure"

d) 8-Queen Problem:

In the Eight Queens problem the challenge is to place eight queens pieces from the game of Chess on a chessboard so that no queen piece is threatening another queen on the board. In the game of chess the queen is a powerful piece and has the ability to attack any other playing piece positioned anywhere else on the same row, column, or diagonals. This makes the challenge quite tricky for humans who often declare after several failed attempts "…there can't be any solution!".

However there are in fact ninety-two valid solutions to the problem although many of those ninety-two are symmetrical mirrors. All of the solutions can be found using a recursive backtracking algorithm. The algorithm works by placing queens on various positions, adding one at a time until either eight queens have been placed on the chess board or less than eight queens are on the board but there are no more safe positions left on the board. When the latter situation is reached the algorithm backtracks and tries another layout of queens.

NOTES: A queen can attack horizontally, vertically, and on both diagonals, so it is pretty hard to place several queens on one board so that they don't attack each other.

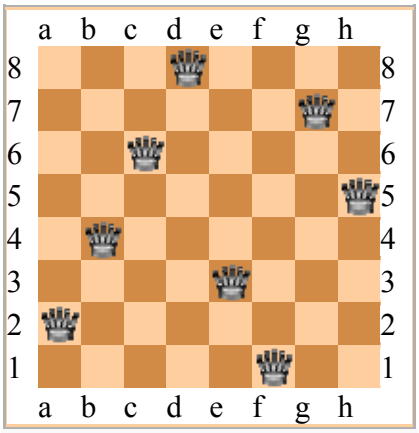


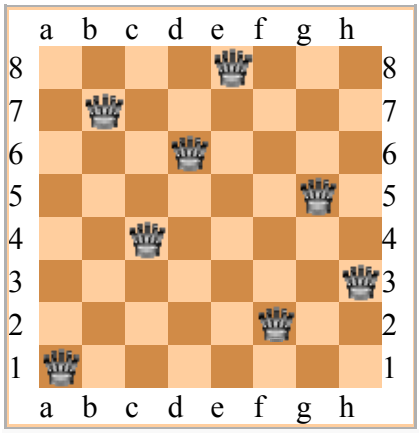
The **n** Queens problem:

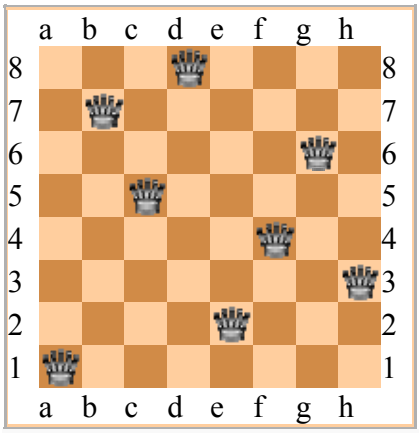
Given is a board of **n** by **n** squares. Is it possible to place **n** queens (that behave exactly like chess queens) on this board, without having any one of them attack any other queen?

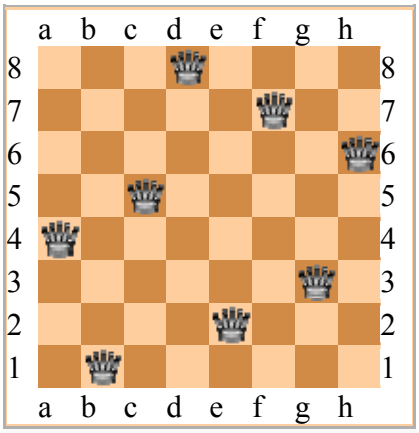
Solutions to the eight queens puzzle

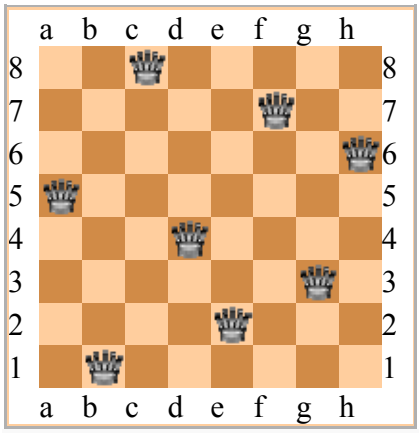
The eight queens puzzle has 92 **distinct** solutions. If solutions that differ only by symmetry operations (rotations and reflections) of the board are counted as one, the puzzle has 12 **unique** (or **fundamental**) solutions, which are presented below:

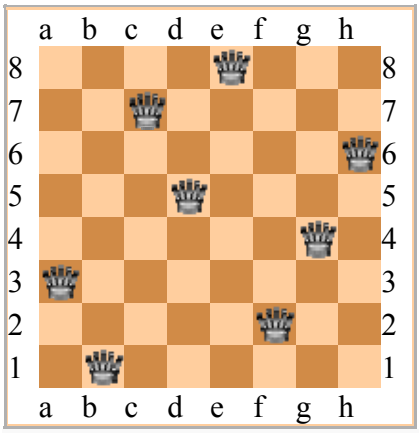

Unique solution 1

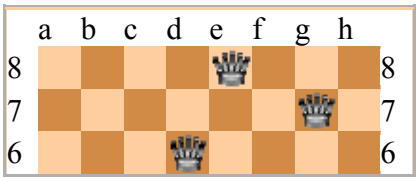

Unique solution 2

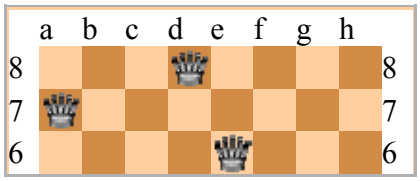

Unique solution 3

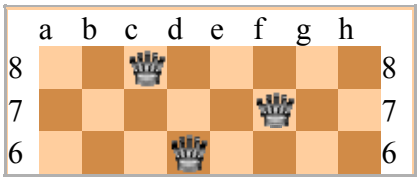

Unique solution 4

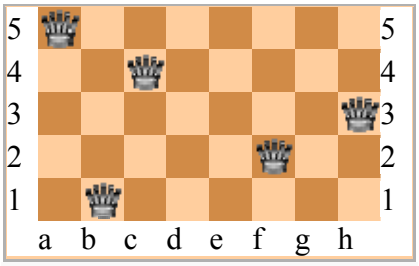

Unique solution 5

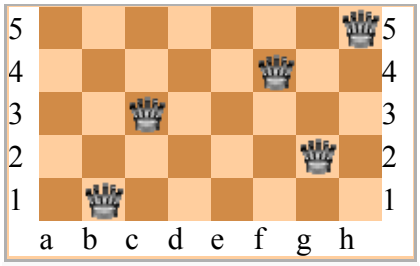

Unique solution 6

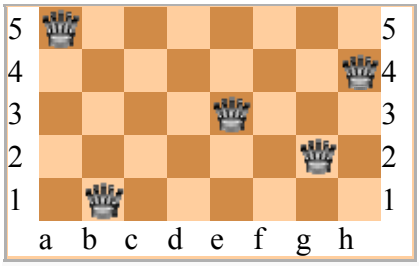


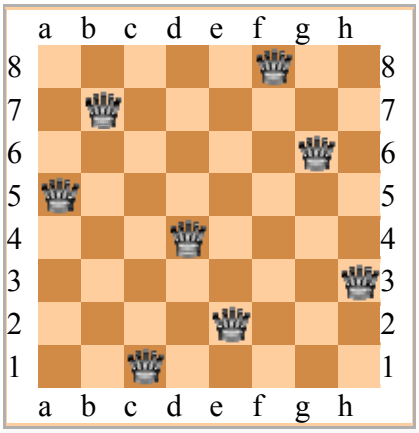


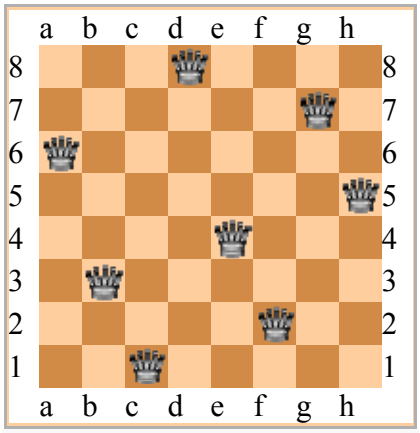


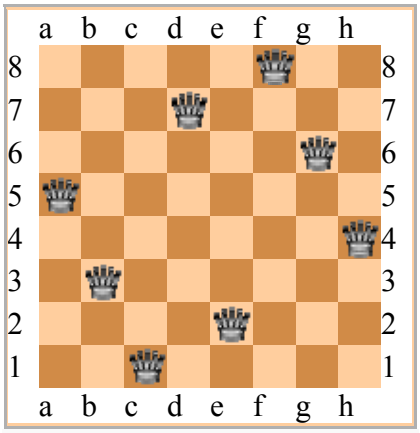

Unique solution 7


Unique solution 8


Unique solution 9


Unique solution 10


Unique solution 11


Unique solution 12

```
Procedure BacktrackingQueenProblem
queen[0, 0, 0, 0, 0, 0, 0, 0 ] { has 8 entries, all have value 0 }
int EightQueen() return array[1…8]
count ← 0
        queen[0] ← 0
        repeat until queen[0] < 8  do
        if is_row_good(0) = YES
        queen[0]← queen[0] + 1
                queen[1] ← 0
                repeat until queen[1] < 8 do
                if is_row_good(0) = YES
                queen[1]← queen[1] + 1
                        queen[2] ← 0
                        repeat until queen[2] < 8 do
                        if is_row_good(0) = YES
                        queen[2]← queen[2] + 1
                                queen[3] ← 0
                                repeat until queen[3] < 8 do
                                if is_row_good(0) = YES
                                queen[3]← queen[3] + 1
                                        queen[4] ← 0
```

```
                                repeat until queen[4] < 8 do
                                if is_row_good(0) = YES
                                queen[4]← queen[4] + 1
                                        queen[5] ← 0
                                        repeat until queen[5] < 8 do
                                        if is_row_good(0) = YES
                                        queen[5]← queen[5] + 1
                                                queen[6] ← 0
                                                repeat until queen[6] < 8 do
                                                if is_row_good(0) = YES
                                                queen[6]← queen[6] + 1
                                                        queen[7] ← 0
                                                        repeat until queen[7] < 8 do
                                                        if is_row_good(0) = YES
                                                        queen[7]← queen[7] + 1
                        count ← count +1
End; //Procedure
```

5.6 Branch and Bound algorithm

Branch and Bound algorithms are similar to backtracking algorithms; however we are looking for an optimal solution. Cost is usually involved; we want to find the lowest cost solution. This algorithm usually grows in a breadth first search manner.

All permutations of a decision are examined, and we follow on the one that has the potential to give the lowest cost or highest valued solutions, if it fails we consider the other decisions.

Optimal Solution for TSP using Branch and Bound

The **Travelling Salesman Problem** (**TSP**) is an NP-hard problem in combinatorial optimization studied in operations research and theoretical computer science. Given a list of cities and their pairwise distances, the task is to find a shortest possible tour that visits each city exactly once.

The problem was first formulated as a mathematical problem in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, a large number of heuristics and exact methods are known, so that some instances with tens of thousands of cities can be solved.

The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept *city* represents, for example, customers, soldering points, or DNA fragments, and the concept *distance* represents travelling times or cost, or a similarity measure between DNA fragments. In many applications, additional constraints such as limited resources or time windows make the problem considerably harder.

In the theory of computational complexity, the decision version of the TSP belongs to the class of NP-complete problems. Thus, it is likely that the worst case running time for any algorithm for the TSP increases exponentially with the number of cities.

The Symmetric Travelling Salesman problem:

A map over the Danish island Bornholm is given together with a distance table showing the distances between major cities/tourist attractions. The problem of a biking tourist, who wants to visit all these major points, is to find a tour of minimum length starting and ending in the same city, and visiting each other city exactly once. Such a tour is called a *Hamilton cycle*. The problem is called the *symmetric* Travelling Salesman problem (TSP) since the table of distances is symmetric. In general a symmetric TSP is given by a symmetric *n x n* matrix *D* of non-negative distances, and the goal is to find a Hamilton tour of minimum length. In terms of graphs, we consider a complete undirected graph with *n* vertices *Kn* and non-negative lengths assigned to the edges, and the goal is to determine a Hamilton tour of minimum length. The problem may also be stated mathematically by using decision variables to describe which edges are to be included in the tour. We introduce 0-1 variables *xij ;* 1 _ *i* < *j* _ *n*, and interpret the value 0 (1 resp.) to mean "not in tour" ("in tour" resp.) The problem is then

$$min \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_{ij}$$

such that

$$\sum_{k=1}^{i-1} x_{ki} + \sum_{k=i+1}^{n} x_{ik} = 2, \qquad i \in \{1, ..., n\}$$

$$\sum_{i,j \in Z} x_{ij} < |Z| \qquad \emptyset \subset Z \subset V$$

$$x_{ij} \in \{0, 1\}, \qquad i, j \in \{1, ..., n\}$$

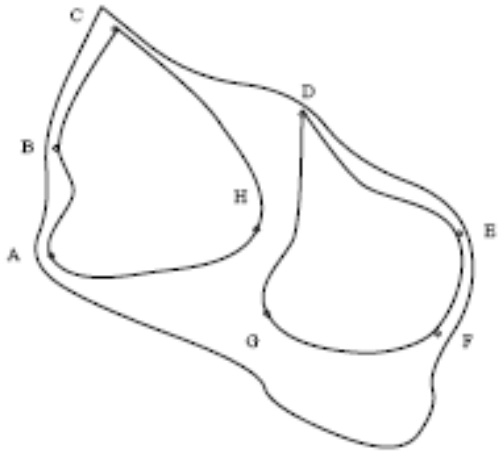


Figure 4: A potential, but not feasible solution to the biking tourist's problem

The first set of constraints ensures that for each *i* exactly two variables corresponding to edges incident with *i* are chosen. Since each edge has two endpoints, this implies that exactly *n* variables are allowed to take the value 1. The second set of constraints consists of the subtour elimination constraints. Each of these states for a specific subset *S* of *V* that the number of edges connecting vertices in *S* has to be less than |*S*| thereby ruling out that these form a subtour. Unfortunately there are exponentially many of these constraints. The given constraints determine the set of feasible solutions *S*. One obvious way of relaxing this to a set of potential solutions is to relax (i.e. discard) the subtour elimination constrains. The set of potential solutions *P* is then the family of all sets of subtours such that each *i* belongs to exactly one of the subtours in each set in the family, cf.

Another possibility is decribed, which in a B&B context turns out to be more appropriate. A subproblem of a given symmetric TSP is constructed by deciding for a subset *A* of the edges of *G* that these must be *included* in the tour to be constructed, while for another subset *B* the edges are *excluded* from the tour. Exclusion of an edge (*i; j*) is usually modeled by setting *cij* to *1*, whereas the inclusion of an edge can be handled in various ways as e.g. graph contraction. The number of feasible solutions to the problem is $(n-1)!/2$, which for $n = 50$ is appr. $3x10^{62}$

5. 7 THE ANALYSIS OF ALGORITHMS

The analysis of algorithms is made considering both qualitative and quantitative aspects to get a solution that is economical in the use of computing and human resources which improves the performance of an algorithm.

A good algorithm usually posses the following qualities and capabilities:

1. They are simple but powerful and general solutions.
2. They are user friendly
3. They can be easily updated.
4. They are correct.

5. They are able to be understood on a number of levels.
6. They are economical in the use of computer time, storage and peripherals.
7. They are well documented.
8. They are independent to run on a particular computer.
9. They can be used as subprocedures for other problems.
10. The solution is pleasing and satisfying to its designer.

Computational Complexity

The computational complexity can be measured in terms of space and time required by an algorithm.

Space Complexity

The space complexity of an algorithm is the amount of memory it needs to run the algorithm.

Time Complexity

The time complexity of an algorithm is the amount of time it needs to run the algorithm. The time taken by the program is the sum of the compile time and run time. To make a quantitative measure of an algorithm's performance, the computational model must capture the essence of the computation while at the same time it must be divorced from any programming language.

Big Oh (O) – Upper Bound:

T(N)

5.8 Asymtotic Notation

Asymtotic notations are method used to estimate and represent the efficiency of an algorithm using simple formula. This can be useful for seperating algorithms that leads to different amounts of work for large inputs.

Comparing or classify functions that ignores constant factors and small inputs is called as asymtotic growth rate or asymtotic order or simply the order of functions. Complexity of an algorithm is usually represented in O, o, , notations.

Big - oh notation (O)

This is a standard notation that has been developed to represent functions which bound the computing time for algorithms and it is used to define the worst case running time of an algorithm and concerned with very large values of N.

Definition : - T(N) = O(f(N)), if there are positive constants C and n_o such that T(N) Cf(N) when N n_o

BIG - OMEGA NOTATION (Ω)

This notation is used to describe the best case running time of algorithms and concerned with very large values of N.

Definition : - T(N) = omega(f(N)), if there are positive constants C and n_o such that T(N) CF(N) when N n_o

BIG - THETA NOTATION

This notation is used to describe the average case running time of algorithms and concerned with very large values of n.

Definition : - T(N) = theta (F(N)), if there are positive constants C_1, C_2 and n_o such that

T(N) = O(F(N)) and T(N) = (F(N)) for all N n_o

Definition : - T(N) = (F(N)), if there are positive constants C_1, C_2 and n_o such that

T(N) = O(F(N)) and T(N) = (F(N)) for all N n_o.

Little - Oh Notation (o)

This notation is used to describe the worstcase analysis of algorithms and concerned with small values of n.

Basic Asymptotic Efficiency Classes

Computing Time Name

0(1) constant

0(log n) Logarithmic function

0(n) Linear

$0(n^2)$ quadratic

$0(n^3)$ cubic

$0(2^n)$ exponential

0(nlogn) n - log - n Logarithmic

n! factorial

WORST - CASE, BEST - CASE AND AVERAGE - CASE EFFICIENCIES

Worst - Case - Efficiency

The worst - case efficiency of an algorithm is its efficiency for the worst - case input of size n, which is an input of size n for which the algorithm runs the longest among all possible inputs of that size

Best - Case Efficiency

The best - case efficiency of an algorithm is its efficiency for the best case input of size n, which is an input of size n for which the algorithm runs the fastest among all possible inputs of that size.

Average - Case Efficiency

The average - case efficiency of an algorithm is its efficiency for the random input of size n, which makes some assumptions about possible inputs of size n.

For example, let us consider sequential search

ALGORITHM

SequentialSearch (A[0...n-1],K)

// Input : An array A[0..n-1] and a search key k.

// Output : Returns the index of the first element of A that matches R or -1 if there are no matching elements.

while i < n and A[i] # k do

i i + 1

if i < n return i

else return - 1

Here, the best - case efficiency is 0(1) where the first element is itself the search element and the worst - case efficiency is 0(n) where the last element is the search element or the search element may not be found in the given array.